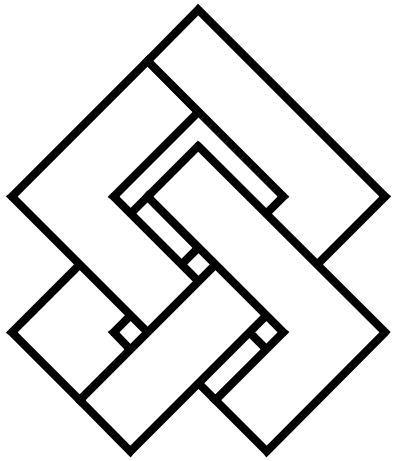

Dumpleton Software Consulting Pty Limited

# OSE

**Version 7.0pl5**

# Python Manual

19 January 2003

# Table of Contents

# Manual Overview

This manual covers the Python wrappers around the OSE C++ class library. The wrappers make available functionality related to the logging system, the real time events system, the service agent framework for creating distributed applications, the HTTP servlet framework and the RPC over HTTP interfaces.

| | |
|---|---|
| Python Modules | Lists the available Python modules and their purpose. Includes brief details regarding installation and setup of the users environment. |
| Logging Facility | Describes the message logging facility, including how to direct messages to a specific log channel, how to log messages to a file or to process them within the actual application. |
| Program Setup | Describes the interface to the configuration database, user environment and other process information. |
| Event Framework | Describes the interface to the event system, how to schedule jobs and setup callbacks in response to real time events such as timers, signals and socket activity. |

| | |
|---|---|
| Service Agents | Describes how to create service agents, add them to groups, subscribe to announcements regarding specific services or membership of specific service groups. |
| Service Reports | Describes how to subscribe to reports published by specific services. Ie., describes the publish/subscribe functionality provided by the service agent framework. |
| Service Requests | Describes how to send requests to remote or local service agents and how to handle any response or error which results. Ie., describes the messaging or request/reply functionality of the service agent framework. |
| Message Exchange | Describes how to connect up processes to form a distributed application, including a decentralised message exchange and exchange groups. |
| Message Encoding | Describes the Python types which can be used in messages and how this can be extended to incorporate new scalar data types. |
| Servlet Framework | Describes the HTTP daemon and servlet framework, including the predefined servlets and how to create customised HTTP server objects. |
| Servlet Objects | Describes how to create new servlet objects including how to handle forms, client congestion and delayed responses. |
| Servlet Plugins | Describes how to create servlet plugins and how to use the Python plugin to dynamically load servlets at runtime from the file system. |
| Remote Access | Describes the RPC over HTTP interfaces into the service agent framework, including support for NET-RPC, XML-RPC and SOAP protocols. |

# Python Modules

OSE includes a number of Python modules. The main module is a wrapper around functionality provided in the OSE C++ class library. Those parts of the OSE C++ class library for which a Python wrapper are provided are the logging system, the real time events system, the service agent framework for creating distributed applications and the HTTP servlet framework.

Additional modules provide access to the OSE service agent framework using an RPC over HTTP protocol called NET-RPC as well as the XML-RPC and SOAP protocol. Note that the XML-RPC and SOAP protocols come with restrictions deriving from problems in the respective protocols and the NET-RPC protocol provides the best integration.

Because interfaces are provided for the OSE service agent framework in both C++ and Python, an application may be spread across multiple processes and consist of processes written using either C++ or Python code. Using shared libraries and dynamic loading, C or C++ code could also be loaded into Python to perform some functions if desired.

Overall, the Python wrappers provide an interface to the functionality of the OSE C++ class library which is easier to use than if the C++ class library were used directly. This makes the Python wrappers ideal for building up the overall structure of a distributed system, with C++ code being used only when necessary.

## Module Descriptions

The Python modules, their names and their purpose are described below.

| Module | Purpose |
|---|---|
| `netsvc` | This is the main module and provides wrappers around the functionality of the OSE C++ class library. It includes all that is required for building distributed applications using the service agent framework. |
| `netrpc` | This module provides a client implementation of the RPC over HTTP protocol implemented by OSE called NET-RPC. |
| `netsvc.xmlrpc` | This module includes the XML-RPC gateway for OSE. Any server code is the same as for when the NET-RPC protocol is used. The only difference is which gateway you instantiate. |
| `netrpc.xmlrpc` | This module provides a client implementation of the XML-RPC protocol. The module is interface compatible with the"`netrpc`" module. |
| `netsvc.soap` | This module includes the SOAP gateway for OSE. Any server code is the same as for when the NET-RPC protocol is used. The only difference is which gateway you instantiate. |
| `netrpc.soap` | This module provides a client implementation of the SOAP protocol. The module is interface compatible with the"`netrpc`" module. |

## Installation and Setup

The "`netsvc`" module requires the main OSE C++ class library to be installed, as well as the Python extension library. The version of "`makeit`" installed when OSE is installed needs to be run in the "`python`" subdirectory of the OSE source code. This final step will install the two Python modules, a dynamically loadable module which drags in the OSE C++ class libraries and a GUI based debugger for the service agent framework called "`spyon`". The exact steps which need to be followed are given in the "`INSTALL`" file in the OSE source code.

When the Python modules are installed, they are not installed into your Python installation, but into the same area that OSE is installed. In order that Python can find the modules, you will need to set your `PYTHONPATH` environment variable to include the appropriate library directory in the OSE installation. For OSE 7.0, if installed into its standard location, the directory will be:

```
/usr/local/ose/7.0/lib/python
```

An OSE installation supports libraries for different architectures. In order that the shared libraries for your specific platform can be found by the Python module, you should ensure that the OSE_HOST variable is set to the same value it was set to when OSE was installed. For example:

```
OSE_HOST=X86_LINUX
```

If you want to be able to run the "spyon" debugger, your PATH environment variable should include the OSE bin directory. For OSE 7.0, if installed into its standard location, the directory will be:

```
/usr/local/ose/7.0/bin
```

If you want to be able to build up a version of the Python wrappers with a DLL for Win32, you have two choices. The first requires you to have access to either the Cygnus Win32 toolkit or MKS toolkit, and the Microsoft C++ compiler. In this case the normal build procedure for OSE is followed. If you only have access to the Microsoft C++ compiler, a native makefile is provided with the source code in the "win32" directory. You should follow the instructions contained in that directory.

Note that if you wish to use either the SOAP client or SOAP gateway, you will need to separately obtain and install the "ZSI" package from the "pywebsvcs" project on SourceForge. The project site address is "http://sourceforge.net/projects/pywebsvcs". You must have version 1.2 RC2 or later of the ZSI package.

## Additional Information

As the main Python module is a wrapper around functionality provided in the OSE C++ class libraries, it may be worthwhile to also consult the manual pages for the corresponding classes in the C++ class library and the general C++ class library manual. The behaviour of some features is controlled using environment variables and not all of these may be mentioned in the manual for the Python modules.

# Logging Facility

The logging facility provides you with a mechanism for generating and capturing messages generated by your application. These can be automatically saved to a log file, or intercepted and dealt with in some other way. The majority of functionality for this feature is provided by the `OTC_Logger` class in the OSE C++ class library.

Some of the features of the logging facility are optional and controlled via environment variables. You should consult the manual page for the `OTC_Logger` class and the general OSE C++ class library manual as a number of these features will not be described here or covered only briefly.

## Logging a Message

The logging facility provides you with the ability to log a message string with a specified priority or level assigned to it. The level is analogous to that used by the UNIX function called "`syslog()`".

| Level | Usage |
|---|---|
| `LOG_EMERGENCY` | A panic condition. |
| `LOG_ALERT` | A condition that should be corrected immediately, such as a corrupted system database. |
| `LOG_CRITICAL` | Critical conditions, such as hard device errors. |
| `LOG_ERROR` | Errors. |
| `LOG_WARNING` | Warning messages. |

| Level | Usage |
|---|---|
| LOG_NOTICE | Conditions that are not error conditions, but that may require special handling. |
| LOG_INFO | Informational messages. |
| LOG_DEBUG | Message that contain information normally of use only when debugging a program. |

To log a message, a handle to an instance of the `Logger` class is acquired and the "`notify()`" member function is called.

```
import netsvc
logger = netsvc.Logger()
logger.notify(netsvc.LOG_DEBUG,"message")
```

The format of a message when displayed will be:

```
DEBUG: message
```

The string before the "`:`" corresponds to the level assigned to the message. The remainder of the line after the "`:`" is the actual message. If you wish to have the time and process ID appear in the prefix, call the "`enableLongFormat()`" member function. Whether the longer form of prefix is enabled can be queried using the "`longFormatEnabled()`" member function. It can be disabled using the "`disableLongFormat()`" member function.

By default, messages will appear on the standard error output. If you wish to disable the display of messages onto the standard error output, call the "`disableStderrOutput()`" member function. Conversely, the "`enableStderrOutput()`" member function can be called to enable display of messages onto the standard error output if previously disabled. Whether messages are currently being displayed onto the standard error output can be queried by calling the member function "`stderrOutputEnabled()`".

## Specifying a Log File

At any time, messages can be captured into a single file by specifying the name of a log file using the member function "`setLogFile()`". If a log file is currently in use, the name of the log file can be queried using the "`logFile()`" member function.

```
logger.setLogFile("/var/tmp/application.log")
```

The string used to specify the name of a log file may incorporate the following special tags.

| Tag | Purpose |
|---|---|
| %h | Will encode the hostname of the machine into the name of the log file. |
| %p | Will encode the process ID into the name of the log file. |
| %Y | Will encode the current year as 4 digits into the name of the log file. |
| %y | Will encode the current year as 2 digits into the name of the log file. |
| %m | Will encode the current month of the year as a zero padded 2 digit number into the name of the log file. |
| %d | Will encode the current day of month as a zero padded 2 digit number into the name of the log file. |

When the tags corresponding to dates are used, a new log file will automatically be created when the value corresponding to a date component changes. The following will for example result in a new log file being created each day.

```
logger.setLogFile("/var/tmp/application-%Y-%m-%d.log")
```

Note that older log files will not be removed automatically, so some other mechanism such as a cron job will need to be employed to remove them.

The name of a log file can also be set using the OTCLIB_LOGFILE environment variable instead of calling "setLogFile()". Similarly, output to the standard error output can be disabled using the OTCLIB_NOLOGSTDERR environment variable and the inclusion of the time and the process ID in the message prefix enabled using the OTCLIB_LOGLONGFORMAT environment variable. If used, these environment variables must be set before the application is run or at least before the "netsvc" module is imported for the first time.

```
import os
os.putenv("OTCLIB_LOGFILE","/var/tmp/application-%Y-%m-%d.log")
import netsvc
```

When an application first attempts to open a log file, if it already exists it will be truncated. If you do not want the log file truncated, but want messages to be appended to an existing log file, the OTCLIB_APPENDLOGFILE environment variable must be set. Again, this needs to be set prior to the application being run or at least before the "netsvc" module is imported for the first time.

Note that if any of these environment variables are used, but calls are subsequently made to the corresponding member functions of the Logger class from within the application, the values of the environment variables will effectively be overridden from that point onwards.

## Specifying a Log Channel

When logging a message, a log channel may also be specified. If the name of a log channel starts with a character other than an alphanumeric character, the message will not be displayed on the standard error output or appear in the log file. If it is displayed or captured in the log file, the name of the log channel does not appear anywhere in the message. The intent of the log channel is to allow one part of an application to capture specific messages produced by another part of the application and deal with them in a special way.

To log a message against a specific log channel, the member function "`notifyChannel()`" is used. The name of the log channel is supplied as the first argument.

```
logger.notifyChannel("VISIBLE",netsvc.LOG_DEBUG,"message")
logger.notifyChannel("#HIDDEN",netsvc.LOG_DEBUG,"message")
```

Messages logged against a specific log channel, can be captured by calling the member function "`monitorChannel()`", supplying the name of the log channel and a callback function.

```
def callback(channel,level,message):
  print (channel,level,message)

logger.monitorChannel("#HIDDEN",callback)
```

The message supplied to the callback function is the original message and does not contain the prefix describing the priority or level assigned to the message, nor does it contain any details relating to the current time or process ID. If you are going to subsequently log the message to a file, you would need to add these details yourself if you require them.

Only one callback can be associated with a particular log channel. If multiple callbacks are required for a particular log channel, separate instances of the `Logger` class should be used. To stop monitoring a specific log channel, the member function "`monitorChannel()`" is called again but with "`None`" supplied in place of the callback function.

If the callback function was a member of a class, it is important to deregister the callback, else a reference to the instance of the class will be maintained and it may not get deleted. You can also deregister all of the callbacks associated with a particular instance of the `Logger` class by calling the member function "`destroyReferences()`". This would be necessary if the class containing the callbacks also held a reference to the instance of the `Logger` class. In this case, a circular reference would exist and neither object would ever be destroyed.

## Logging Python Exceptions

To make the task of logging details of a Python exception easier, the "`logException()`" function is provided by the "`netsvc`" module. This function should only be called from within the context of a Python "`except`" clause. The information logged is similar to that displayed by Python when an exception is not caught and includes details of the exception and a stack trace.

```
try:
  function()
except SystemExit:
  raise
except:
  netsvc.logException()
  sys.exit()
```

The details of the exception are logged with level "`LOG_ERROR`" and a specific log channel is not specified. If you wanted to log the details of the exception to a specific log channel, or vary the level, you can use the "`exceptionDetails()`" function of the "`netsvc`" module to obtain the same information that would be logged by the "`logException()`" function and then call the "`notify()`" member function of an instance of the `Logger` class yourself.

```
try:
  function()
except SystemExit:
  raise
except:
  details = netsvc.exceptionDetails()
  logger.notifyChannel("WARNING",netsvc.LOG_WARNING,details)
  pass
```

If you don't want the stack trace and only want the description of the exception, use the function "`exceptionDescription()`" instead. The result of calling either of these functions need not be used with the logger, but could be displayed using any other available mechanism as well.

Note that the "`exceptionDetails()`" and "`exceptionDescription()`" functions are also available in the "`netrpc`" module if you are using that in a standalone client application.

## Exceptions in a Callback

Whenever a callback is executed, it occurs as a result of a call from C++ code into Python code. Because of the mix of C++ code and Python code, if an exception occurs within the callback function, Python can't by itself properly shutdown the application. This is further complicated by the fact that a callback can be called within the context of a callback from the event dispatcher.

As a consequence, when any callback into Python code from C++ occurs, if a Python exception occurs and the callback itself doesn't catch it and deal with it, it will be caught with the details of the exception being logged. The event dispatcher will then be stopped if it is running and the "`SystemExit`" exception raised in order to prevent Python from running any further code. The outcome is the same as when only Python code is being used, except that the details of the exception are displayed using the logging facility rather than being dumped directly onto the standard error output.

# Program Setup

As Python is an interpreted language, configuration of an application can be carried out by editing the actual scripts. In some circumstances however, it is still easier or more practical to rely upon a configuration database or environment variables. When using OSE this is especially the case, as an application can be a mix of C++ and Python code and configuration data may need to be accessible from code written in both languages.

To support this the Python wrappers provide an interface to the configuration database of the OSE C++ class library. The corresponding class in the OSE C++ class library which provides this functionality is the `OTC_Program` class. Not all functionality of this class is mirrored in the Python interface as Python has its own way of doing most of what is provided by this class. Access is however provided to aspects of the configuration database and environment variable database. The functionality for generating unique identifiers is also exposed.

## Configuration Database

The configuration database is an in memory database. The database may be populated by calls from within the application, or by loading in a configuration file. The configuration database may also be saved to a file. In essence, the configuration database is not much more than a dictionary mapping names to values.

To initially load the configuration database from a file, the "`loadConfig()`" function is used. A single configuration item may be explicitly merged into the configuration database using the "`mergeConfig()`" function. A query can subsequently be made against the configuration database using the "`lookupConfig()`" function. If no match is found in the configuration database for the item in question, the value `None` is returned.

```
import netsvc
import os

netsvc.loadConfig("database.cfg")
netsvc.mergeConfig("PWD",os.getcwd())

print netsvc.lookupConfig("PWD")
```

A single configuration item can be removed from the database using the "`removeConfig()`" function. The configuration database can be completely emptied using the function "`removeAllConfig()`". The contents of the configuration database can be saved to a file using the "`saveConfig()`" function.

```
netsvc.removeConfig("PWD")
netsvc.saveConfig("database.cfg")
```

## Configuration File

The only real restrictions in regard to naming is that the colon character should not be used anywhere in a name, a name should not being with an exclamation mark and whitespace should not be used at the start or end of a name. The colon character cannot be used as it used in a configuration file to separate the name from the value. A leading exclamation mark should not be used as it is used to denote a comment.

If these characters are used in a name and the configuration database is saved to a file, the results when that configuration file is read back in will not be the same. The only other special character when used in a configuration file is a back slash, which when used at the end of the line, indicates the following line is part of the same value. Note that the leading whitespace and the whitespace either side of the colon will be ignored when the configuration file is read in.

```
! comment
single-line-value : value
multi-line-value : value\
  value
```

When reading in a configuration file using "`loadConfig()`", an exception is raised only if the file doesn't exist or the file couldn't be opened. If there are no errors in the file, the value `None` is returned. If there are errors in the file, a string is returned which contains details of the errors and what action has been taken. By default, the details of the errors are also output via the logging system on the default log channel.

If details of any errors should be output on a specific log channel, an optional second argument can be supplied to the "`loadConfig()`" function giving the name of the log channel. If the value `None` is supplied in place of the name of the log channel, the details of the errors will not be output via the logger at all. The value `None` could be used if you wish to amend the details of the errors before they are logged.

```
file = "database.cfg"
errors = netsvc.loadConfig(file,None)
if errors:
  errors = "Error reading %s\n%s" % (‘file‘,errors)
  netsvc.Logger().notify(netsvc.LOG_DEBUG,errors)
```

## Naming Hierarchies

If a naming hierarchy is required, the components of the hierarchy within the name should be separated by using a period.

```
compiler.preprocessor.debug-level : 0
compiler.parser.debug-level : 1
compiler.code-generator.debug-level : 0
compiler.assembler.debug-level : 0
```

In general, the purpose of using a naming hierarchy is to associate properties with the same name with different parts of an application, or with different instances of some object. To cater for default values, rather than enumerating all possible objects, a wildcard can be specified in place of a single component in a naming hierarchy. This says to match any component name in this position. Only those items which need to be different then need to be explicitly specified.

```
compiler.*.debug-level : 0
compiler.parser.debug-level : 1
```

When a lookup is made against the database, a check is first made for any entry which matches exactly the name of interest. If this name is not present, a search is then made of the entries containing a wildcard. If a match is found, the value associated with the wildcard entry will be returned. If there are multiple wildcard entries which match a lookup against the configuration database, that which has the longest leading exact match will be used.

## Environment Variables

In addition to the configuration database, an interface is also provided to the standard operating system environment variables. Python does already provide an interface for this, however the Python interface does have a few quirks which can sometimes make it less than useful.

One problem with the standard Python interface is that when "`os.putenv()`" is used to set an environment variable, that variable is not then visible using "`os.getenv()`". This is because "`os.getenv()`" uses "`os.environ`", which is a copy of the environment which is populated at startup and any changes to environment variables are not reflected in that copy.

As such, although changes to the environment will be seen by subprocesses, they will not be visible in the same process. This means that an environment variable can’t at the same time be used to transfer information to a different part of the application.

To lookup the value of an environment variable the function "`lookupEnviron()`" is used. If a new environment variable needs to be set, or an existing value changed, the function "`mergeEnviron()`" is used. Any changes to the environment variables will be visible immediately, but there is no way to get a list of all environment variables which are set. When a lookup is made but no such environment variable exists, the value `None` is returned.

```
netsvc.mergeEnviron("PWD",os.getcwd())
print netsvc.lookupEnviron("PWD")
```

In addition to these functions, the function "`expandEnviron()`" is provided. This function accepts a string and replaces any reference to an environment variable specified using Bourne shell syntax, with that environment variables actual value. The intent in providing this function is that it can be used in conjunction with the configuration database, allowing configuration items to refer to environment variables.

```
application.log-files : ${HOME}/logs
```

Note that the expansion isn't automatic when a lookup is made against the configuration database. The application code will have to explicitly expand the value obtained form the configuration database.

```
value = netsvc.lookupConfig("application.log-files")
directory = netsvc.expandEnviron(value)
```

## Unique Identifiers

In many applications, it is often useful to be able to create abstract identifiers to uniquely identify objects or resources. These might be used to identify user sessions in a web based application, specific requests in a distributed messaging system, or even the particular service agent which a request in a distributed messaging system is targeted at.

Such identifiers may only need to be unique within the context of the lifetime of the application, or possibly may need to be globally unique. In the case of the latter, to be rigourous this would normally require an external database to be maintained which tracks what identifiers have been used. In most cases however, it is not necessary to go to that extent and a simplistic means can be used to generate a psuedo unique identifier which is sufficient.

To generate such identifiers the function "`uniqueId()`" is provided. The function can provide identifiers in either a short or long format. In the short format, the identifier contains components which identify the host on which the process is running, the process id and an incremental counter. In the long format, time values are also included which tie the identifier to an instant in time.

```
id1 = netsvc.uniqueId(netsvc.ID_SHORT_FORMAT)
id2 = netsvc.uniqueId(netsvc.ID_LONG_FORMAT)
```

If you wish to incorporate your own prefix into the identifier, an optional second argument can be supplied to the "`uniqueId()`" function.

```
id1 = netsvc.uniqueId(netsvc.ID_SHORT_FORMAT,"$SID?")
```

The short format identifier is suitable for use within the context of a single process. Duplicates would only be encounterd if the incremental count of the number of identifiers exceeded what can be stored within a 32 bit integer value. If this were to occur, the counter would wrap around to zero and conflicts might thus arise if the existing identifier were still active.

The short format identifier could also be used within the context of a constrained distributed application provided that the nature of the application is such that knowlege of what the identifier is associated with is always discarded when the process the identifier is bound to is destroyed. This would be necessary, as the identifier could be reused if the process id was reused at some latter point.

If a better gaurantee of uniqueness over time is required, the long format identifier should be used. In this case, the identifier also records the time at which the first identifier was generated by the process, as well as a time delta as to when that particular identifier was generated. Incorporation of time information avoids problems with the incremental counter overflowing and reuse of the same process id at a latter point in time.

## Process Identity

A further feature which is useful in distributed applications is a way of identifying specific processes. Such an identifier can be generated by combining the name of the host and the process id into a single string. To facilitate this, the function "`processIdentity()`" is provided.

```
identity = netsvc.processIdentity()
```

# Event Framework

The main support for concurrency in the OSE C++ class libraries comes in the form of a mechanism for building event driven systems. This is based around a central job queue and a dispatcher, which takes successive jobs from the queue and executes them. To support real time systems, there also exist a number of event sources which will schedule jobs to trigger an agent to be notified when an event of interest occurs. The major event sources include timers, signals and the availability of data for reading on a socket.

The major classes in the OSE C++ class library involved in providing this functionality are the `OTC_Dispatcher`, `OTC_EVAgent` and `OTC_Job` classes, plus the various event classes related to the event sources. In the C++ implementation, communication of events is mainly performed by passing around event objects and having a single event handler method in an agent to deal with them. In the Python implementation, separate callback functions can be registered by an agent against each event of interest.

Note that only the major features of the C++ implementation are reflected in the Python interface. Python does not provide a means of creating your own event types or event sources. A Python agent is also not able to process any events except those from the major event sources.

## Scheduling a Job

Scheduling of jobs comes in the form of registering a callback function with the dispatcher for execution. A job may be scheduled as a priority job, a standard job, or an idle job. The type of job determines where in the order of existing jobs, a new job will be placed. Any priority jobs are executed before a standard job is processed. When there are no priority jobs or standard jobs remaining, any pending idle

jobs will be reclassified as standard jobs and subsequently executed. When scheduling a job, if jobs of the same type already exist, the new job will be placed at the end of the list of jobs of the same type.

To schedule a job the dispatcher member function "`schedule()`" must be called, supplying the callback function and the type of job. To set the dispatcher running, the member function "`run()`" is called. If the only feature of the event system which is used is that of scheduling jobs, the "`run()`" function will return when there are no more jobs to execute. A job may prematurely stop the dispatcher by calling the "`stop()`" member function. If a callback raises an exception which is not caught and processed within the callback itself, the details of the exception will be logged, the dispatcher stopped and Python exited immediately.

```
def callback(message="hi"):
  print message

dispatcher = netsvc.Dispatcher()
dispatcher.schedule(callback,netsvc.IDLE_JOB)
dispatcher.schedule(callback,netsvc.STANDARD_JOB)
dispatcher.schedule(callback,netsvc.PRIORITY_JOB)
dispatcher.run()
```

The callback supplied when scheduling a job can be a normal function or a member function associated with an instance of a class. If a callback function is scheduled directly with the dispatcher in this way, it will be called with no arguments and cannot be cancelled once scheduled.

If it is necessary to pass arguments to a callback function, an instance of the `Job` class must be used in place of the actual callback function. The `Job` class will hold a reference to the real callback function as well as the arguments. When the job is executed it will call the callback function with the supplied arguments.

```
job = netsvc.Job(callback,("bye",))
dispatcher.schedule(job,netsvc.IDLE_JOB)
```

In addition to providing a means of supplying arguments to a callback function, the `Job` class provides a means of cancelling execution of a callback function. In order to do this, a reference to the instance of the `Job` class should be kept. If it is subsequently necessary to cancel execution of the callback prior to it having being called, the "`cancel()`" member function of the `Job` class should be called.

```
job = None
def callback1():
  print "hi"
  job.cancel()
def callback2():
  print "hi"

dispatcher.schedule(callback1,netsvc.PRIORITY_JOB)
job = netsvc.Job(callback2)
dispatcher.schedule(job,netsvc.STANDARD_JOB)
```

All that is occuring here is that when the "`cancel()`" member function is called, a flag is set. When the job is executed it will note that the flag is set and will not execute the callback function. If the callback function is a member function of a class, it is important to ensure that any reference to the instance of the `Job` class is destroyed when no longer required. If this is not done and the reference is a member variable of the same class the callback function is a member of, a circular reference will exist and that instance of the class will not be able to be destroyed.

Any arguments to be passed to the callback function would by default be supplied when the instance of the `Job` class is created. If it is necessary to generate an instance of the `Job` class such that it can be passed to another part of the program, but the arguments to the callback function are not known at that time, it is instead possible to supply the arguments at the time the job is scheduled. This is done by using the "`schedule()`" member function of the `Job` class rather than that of the dispatcher. Any arguments supplied in this way will override those provided when the instance of the `Job` class is created.

```
job = None
def callback1(message):
  print message
  job.schedule(netsvc.STANDARD_JOB,("override",))
def callback2(message)
  print message

job = netsvc.Job(callback1,("default",))
job.schedule(netsvc.STANDARD_JOB)
job = netsvc.Job(callback2)
```

This would allow for instance a class which accepts callback registrations to return a reference to a `Job` class which will later be used to schedule the callback with an as yet undetermined set of arguments. The client who registered the callback could however cancel execution of the callback before it is called.

Once "`cancel()`" has been called on an instance of a `Job` class, whether or not it has already been scheduled, the callback function will never be executed. To reset the flag which makes the callback function runnable, the "`reset()`" member function should be called. To determine if the an instance of the Job class is still in a runnable state, a truth test can be performed on it.

```
if job:
  # job wasn't cancelled
  job.schedule(netsvc.STANDARD_JOB)
else:
  # job was cancelled
  pass
```

If you wish to use the `Job` class separate to the dispatcher, you can trigger execution of the callback function by calling the "`execute()`" member function. If any arguments are supplied to the "`exe-`

cute()" member function, these will override any which may have been supplied when that instance of the Job class was created.

## Real Time Events

The Python interface provides the ability to register interest in a number of real time events. These are program shutdown, one off alarms or actions, recurring actions, timers, signals and data activity on sockets. That an event of interest has occurred is notified by execution of a callback supplied at the time that interest in an event is registered.

In the C++ implementation, the methods for expressing interest in a specific type of event were spread across numerous classes. In the Python interface, all functions for registration of interest in events are contained within the Agent base class. Any object interested in receiving notification of an event occurring is expected to derive from the Agent class.

The simplest type of notification isn't really a real time event at all, but a variation on the concept of scheduling a job with the dispatcher. Instead of calling the "schedule()" member function of the dispatcher, the "scheduleAction()" member function of the Agent base class is called.

The major difference between using "scheduleAction()" and "schedule()" is that when using "scheduleAction()" you can optionally supply an additional string argument to be used as an identifier for that job. This identifier can be used to cancel the job before it actually gets executed by calling "cancelAction()". If the callback funcion accepts a single argument, the identifier will also be passed to the callback function as argument. The identifier can thus be used to distinguish between different jobs calling the same callback function. If an identifier is not explicitly provided, a unique internal identifier will be created. Whether or not the identifier is set explicitly or created internally, the identifier used is returned as the result of the "scheduleAction()" method.

```
class Object(netsvc.Agent):
  def __init__(self):
    self.scheduleAction(self.callback1,netsvc.PRIORITY_JOB)
  def callback1(self):
    self.scheduleAction(self.callback2,netsvc.IDLE_JOB,"hi")
    self.scheduleAction(self.callback2,netsvc.IDLE_JOB,"bye")
  def callback2(self,name):
    print name
    if name == "hi":
      self.cancelAction("bye")

dispatcher = netsvc.Dispatcher()
object = Object()
dispatcher.run()
```

When using the Agent class, you still need to run the dispatcher. You do not need to schedule any jobs directly with the dispactcher, but any initial agents need to be created prior to the dispatcher being run. Note that in scheduling a job with a particular identifier, any job already scheduled with that agent

using the same identifier will first be cancelled. If you want to cancel all jobs scheduled using the "`scheduleAction()`" member function you should call the "`cancelAllActions()`" member function.

## Destroying Agents

Ensuring that any outstanding job is cancelled, or deregistering interest in any event source, is important if you are endeavouring to destroy an agent object. If registrations are not cancelled, a circular reference will exist between data held by the instance of the `Agent` base class and the derived object. Such circular references defeat the Python reference counting mechanism, meaning that the object may never be destroyed.

To combat this particular situation, the member function "`destroyReferences()`" is included in the `Agent` base class. This will cancel all outstanding jobs and cancel any interest in other event sources as well, destroying any circular references in the process. Provided there are no other references to the object elsewhere, Python should now be able to destroy it.

If you have circular references within your derived class, you may wish to extend this method in your own class so as to undo those circular references. Using the same member function name will make it less confusing to a user of your class as they will only have to call one function. If this is done, you should ensure however that the last thing the derived version of the method does is call the version of the method in the immediate base class.

## Alarms and Timers

Alarms and timers are a means of having a callback function executed at some point of time in the future. The difference between an alarm and a timer is that an alarm is defined by an absolute value or point in time, where as a timer is defined by a relative offset in time. For an alarm this means supplying the clock time in seconds at which the callback should be executed. For a timer this means supplying the number of seconds from now at which point the callback should be executed.

```
class Object(netsvc.Agent):
  def __init__(self):
    offset = 60
    now = time.time()
    then = now + offset
    self.setAlarm(self.callback1,then)
    self.startTimer(self.callback2,offset,"timeout-1")
    self.startTimer(self.callback2,offset+10,"timeout-2")
  def callback1(self):
    print "alarm"
  def callback2(self,name):
    print name
    if name == "timeout-1":
      self.cancelTimer("timeout-2")
```

The member function for setting an alarm is "`setAlarm()`" and that for starting a timer is "`startTimer()`". The first argument is the callback function, the second argument is the absolute or relative time and the third argument is an optional identifier for that alarm or timer. Scheduling an alarm or timer with an identifier matching that of an alarm or timer which hasn't yet expired will cause that unexpired alarm or timer to be cancelled.

Both types of events are one off events, with the registration being cancelled once the callback has been executed. The identifier may also be used to cancel an alarm or timer before it expires. To cancel an alarm use "`cancelAlarm()`" and to cancel a timer use "`cancelTimer()`". To cancel all pending alarms use "`cancelAllAlarms()`" and to cancel all pending timers use "`cancelAllTimers()`". If an identifier is not excplicitly provided, an internal identifier will be automatically created with it being returned as the result of the function being called to schedule the callback.

## Recurring Actions

A recurring action is where a job is run at regular intervals. Precisely when the callback function associated with a job is executed is determined by a specification of the form used by the UNIX cron utility. The specification consists of five fields each separated by white space. The fields specify:

- minute (0-59),
- hour (0-23),
- day of the month (1-31),
- month of the year (1-12),
- day of the week (0-6 with 0=Sunday).

A field may be an asterisk "`*`", which always stands for "`first-last`". Ranges of numbers are allowed. Ranges are two numbers separated with a hyphen. The specified range is inclusive. For example, `8-11` for an "hours" entry specifies execution at hours `8`, `9`, `10` and `11`.

Lists are allowed. A list is a set of numbers (or ranges) separated by commas. For example, "`1,2,5,9`" and "`0-4,8-12`". Step values can be used in conjunction with ranges. Following a range with "`/number`" specifies skips of the number's value through the range. For example, "`0-23/2`" can be used in the hours field to specify the callback function be executed every other hour. Steps are also permitted after an asterisk, so if you want to say "every two hours", just use "`*/2`".

Names can also be used for the "month" and "day of week" fields. Use the first three letters of the particular day or month (lower case, or first letter only uppercase).

The day that a callback function is to be executed can be specified by two fields, day of month and day of week. If both fields are restricted (ie., aren't "`*`"), the callback function will be executed when either field matches the current time. For example, "`30 4 1,15 * 5`" would cause the callback function to be executed at 4:30 am on the 1st and 15th of each month, plus every Friday.

To schedule this type of job, the "`scheduleAction()`" function is used except that instead of specifying the job type as the second argument, the specification string should be used.

```
class Object(netsvc.Agent):
  def __init__(self):
    self.scheduleAction(self.daily,"0 0 * * *","daily")
    self.scheduleAction(self.weekly,"0 0 * * Sat","weekly")
    self.scheduleAction(self.monthly,"0 0 1 * *","monthly")
    self.scheduleAction(self.yearly,"0 0 1 Jan *","yearly")
    self.scheduleAction(self.holiday,"0 0 25 Dec *","christmas")
  def daily(self):
    print "daily"
  def weekly(self):
    print "weekly"
  def monthly(self):
    print "monthly"
  def yearly(self):
    print "yearly"
  def holiday(self,name):
    print name
```

As a recurring action by nature will always run at some point in the future, you have to explicitly call "`cancelAction()`" to stop it from running, even if it has already run at some point in time already. If you make an error in the specification string such that it is invalid, no indication will be given and the job will simply never be executed. The "`cancelAllActions()`" member function, as well as cancelling actions associated with a once off call of a callback function, will also cancel all recurring actions.

## Socket Events

In an event driven system, it is important that any callback not unnecessarily block waiting for something to happen. If a callback does block, it prevents any other part of the system from doing something. The main reason which a callback may block is due to an attempt to read data from a socket when there is no data waiting to be read. In an event driven system, an application should register interest in the availability of data on a socket and only attempt to read data from the socket when it is known that there is some available.

It is also advantageous in a event driven system for sockets to be placed into non blocking mode. When a socket is in non blocking mode, if data is written to a socket and the socket is full an error is returned indicating that the call would have blocked. The code can now register interest in the possibility of being able to write data to a socket and subsequently be notified when such a call would be successful. In the mean time, other parts of the system can still do something.

To register interest in either of these events, the member function "`subscribeSocket()`" should be used. The first argument to the function should be the callback function, the second argument the socket descriptor and the third argument the type of events. If the third argument is not supplied, it will default to `SOCKET_POLLIN`, indicating interest in the availability of data on a socket for reading.

Other possible values for the third argument are SOCKET_POLLOUT and SOCKET_POLLPRI. The value SOCKET_POLLPRI is similar to SOCKET_POLLIN except that it relates to there being priority out of band data being available for reading. Out of band data is not a feature which is used much these days and isn't implemented the same on all systems. It is probably best to avoid using out of band data.

A final value of SOCKET_POLLOUT indicates interest in when data can be safely written to the socket without the call blocking. Note that this will generally nearly always be the case, so you should only subscribe to this event on a socket, when you know that writing to the socket would cause it to block. Once you have been notified that it is safe to write to a socket and you have written your data, you should immediately unsubscribe to this event on a socket, otherwise your callback will continually be called.

```
class Agent(netsvc.Agent):
  def __init__(self,host,port):
    netsvc.Agent.__init__(self)
    self._host = host
    self._port = port
    self.scheduleAction(self.connect,netsvc.STANDARD_JOB)
  def connect(self):
    self._sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    try:
      self._sock.connect((host,port))
    except:
      dispatcher.stop()
    else:
      self.subscribeSocket(self.read,self._sock.fileno())
  def read(self,fileno,event):
    if fileno != self._sock.fileno():
      return
    if event == netsvc.SOCKET_POLLIN:
      data = self._sock.recv(1024)
      if len(data) == 0:
        self.unsubscribeSocket(self._sock.fileno())
        self._sock.close()
        dispatcher.stop()
      else:
        sys.stdout.write(data)
```

When you are no longer interested in a particular event on a socket, you can unsubscribe to that event using the "unsubscribeSocket()" member function. If called with only a single argument, all events currently of interest on that socket will be unsubscribed. To unsubscribe to only a specific event type, pass the type of event as the second argument.

## Program Signals

The most common circumstance in which an application may receive a program signal is when it is being killed as result of a user interrupting it by typing control-C, or if running UNIX, when the oper-

ating system is being shutdown. Other uses for program signals are to force an application to reread a configuration file.

These three cases are typically indicated by the program signals `SIGINT`, `SIGTERM` and `SIGHUP`. A robust application should at least catch the first two of these signals and cause the program to shutdown gracefully. This may entail ensuring that any data is written out to files, removal of file locks, closing off of database connections etc.

To subscribe to a signal, the member function "`subscribeSignal()`" should be used. The first argument should be a callback function to be called when a particular signal occurs and the second argument the particular signal of interest. A particular agent may only supply one callback for any particular signal, but different agents may subscribe to the same signal with both being notified when it occurs. Although an interest in such a signal is usually persistent, it is possible to unsubscribe from a particular signal using the member function "`unsubscribeSignal()`" and unsubscribe from all signals using "`unsubscribeAllSignals()`".

```
class Agent(netsvc.Agent):
  def __init__(self):
    netsvc.Agent.__init__(self)
    self.subscribeSignal(self.signal,signal.SIGINT)
    self.subscribeSignal(self.signal,signal.SIGTERM)
  def signal(self,signum):
    self.scheduleAction(self.stop,netsvc.PRIORITY_JOB)
  def stop(self):
    netsvc.Dispatcher().stop()
```

In practice, only one of the agents subscribed to `SIGINT` and `SIGTERM` should actually shutdown the dispatcher. This agent should however, not shutdown the dispatcher immediately as other agents may not yet have received their notification that the signal occurred. The agent should instead schedule a priority job to actually stop the dispatcher. This priority job will only be executed after all outstanding signal notifications have been delivered.

## Program Shutdown

Subscription to a program signal provides a means of immediately shutting down an application when caused to do so by an external signal. What program signals don’t do however, is provide a means of initiating a graceful shutdown of an application from within the application itself. An application could send itself a signal, however, this isn’t necessarily practical.

A further problem is that in an event driven system, it may not always be possible to perform everything that is required in a single callback function. What is instead needed is the ability to run the application for a further finite amount of time so that any outstanding operations can be finalised first. At the end of that time, then the application can be stopped.

To support this slightly more orderly mechanism for program shutdown, the member function "`scheduleShutdown()`" is provided. When an agent wishes to force the program to shutdown it

should call this member function. This member function can also be called when an external signal intended to shutdown the program is received. Doing this in the latter case means you don't need to have separate code for the two different cases.

If an agent is interested in the fact that the program is being shutdown, it can call the "`subscribeShutdown()`" member function, supplying a callback function to be called when such an event does occur. Note that the call to "`scheduleShutdown()`" will result in the dispatcher being stopped automatically, so you do not need to do it explicitly. If necessary, an agent can unsubscribe from program shutdown notifications by calling the member function "`unsubscribeShutdown()`".

```
class Agent(netsvc.Agent):
  def __init__(self):
    self.subscribeShutdown(self.shutdown)
    self.subscribeSignal(self.signal,signal.SIGINT)
    self.subscribeSignal(self.signal,signal.SIGTERM)
    self.startTimer(self.timeout,60)
  def timeout(self):
    self.scheduleShutdown()
  def signal(self,signum):
    self.scheduleShutdown()
  def shutdown(self,category):
    if category == netsvc.SHUTDOWN_PENDING:
      # shutdown is pending
    else:
      # shutdown has arrived
```

When shutdown is initiated, any callback function supplied by an agent will actually be called twice. The first time it is called, it will be called with the value "`SHUTDOWN_PENDING`". Once all subscribed agents have been notified that shutdown is pending, the callback function will then be subsequently called again, this time with the value "`SHUTDOWN_ARRIVED`". Upon all agents receiving the second notification, the dispatcher will be stopped and the process will exit.

Note that the second of these notifications will not occur immediately after the first. Exactly how much time may pass is dependent on a number of factors. The first determining factor is the argument supplied to the "`scheduleShutdown()`" member function. If no argument is supplied, or a value of "`0`" is supplied, there will be an inbuilt delay of 1 second between shutdown being scheduled and the program actually being shutdown.

This implicit delay gives scope for activities which can't be factored into a single callback function time to be carried out. For example, it may be necessary to send data via a socket to some remote host and wait for the response. If the default value of 1 second is insufficient, or is too long a time, it can be overridden in a number of ways.

The first way of overriding the default value of 1 second is by setting the environment variable `OTCLIB_SHUTDOWNDELAY`. If this is done, it should be set to a value representing the number of milliseconds to wait. An alternative is to modify each call of "`scheduleShutdown()`" and explic-

itly provide the time delay as an argument. If this is done, the argument should express the number of full or partial seconds as a floating point value.

Using a time delay is a useful starting point, as it provides a means of defining an upper bound on the amount of time you wish to allow the system to run before it is stopped. Having a small delay and ensuring everything is done in that time is preferable, as in certain circumstances such as the operating system sending a `SIGTERM` to an application on system shutdown, the operating system will usually forcibly shutdown your application using `SIGKILL` after 5 seconds if it doesn't do so of its own accord.

Although getting away from the goal of having only one mechanism for shutting down a program, in this circumstance, it may still be preferable to separately identify a `SIGTERM` signal and deal with it differently. Here you might only do anything that is absolutely essential and stop the process immediately. What is the best approach will depend on the specific application in question.

If the problem of a `SIGTERM` signal is ignored, a further mechanism for delaying actual shutdown of a process is also provided. If upon receiving notification of a pending shutdown, an agent knows it needs to wait for some event to occur first, it can call the "`suspendShutdown()`" member function. If this is done, although the shutdown delay may expire, actual program shutdown will not occur until a corresponding call to the "`resumeShutdown()`" member function. If more than one agent calls "`suspendShutdown()`", actual shutdown will not occur until "`resumeShutdown()`" has been called a matching number of times.

Although it is possible to suspend the shutdown process in this way, it is not possible to cancel it completely. But then, if an agent doesn't call "`resumeShutdown()`" at some point it would never actually occur. This wouldn't be very useful however, as more than likely parts of the application may have placed themselves into a dormant state.

Finally, as scheduling program shutdown upon a signal occurring would be done in practically all programs, support for this has been factored into the actual dispatcher. Thus, instead of dedicating a specific agent to catch any signals, the main program file can contain:

```
dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)
dispatcher.monitor(signal.SIGTERM)
```

If this interface is used however, the only means of overriding the delay between shutdown being scheduled and actual shutdown is by the `OTCLIB_SHUTDOWNDELAY` environment variable.

The dispatcher also provides the member function "`shutdown()`". This behaves much the same as the "`scheduleShutdown()`" member function of the `Agent` class. The presence of the "`shutdown()`" member function in the dispatcher, allows code which is distinct from an agent to also schedule a program shutdown.

Note that whatever mechanism is used to initiate program shutdown using these features, messages will be displayed via the logger indicating that shutdown has been scheduled and that it has arrived. Additional messages will be displayed via the logger when the shutdown process is suspended and resumed.

# Service Agents

The service agent framework in OSE provides request/reply and publish/subscribe features similar to that found in message oriented middleware packages. Unlike most of the available packages, the service agent framework does not have a flat namespace with respect to naming, but uses an object oriented model, with each service having its own namespace with respect to subject names for subscriptions and request method names.

Building on this object oriented approach, it is possible to subscribe to the existence of specific services, or to groups of services as well as aspects of the services themselves. By using subscription to groups, an application can be setup to dynamically handle the introduction and withdrawal of new services rather than being hardwired. Services are also able to monitor when subscriptions occur and identify who is making the subscription if necessary.

All the features of the service agent framework can be applied within the scope of a single process, or across a group of distributed processes. A specific service need not even be aware that a service it makes use of is in a remote process as the interface and means of interacting with that service are the same. Services may therefore be moved around between processes or onto different machines and the key parts of the application will not need to be changed.

As the Python interface is simply a wrapper on top of functionality provided by the OSE C++ class library, you are not restricted to writing service agents in just Python. In a distributed application for example, one process may be entirely written in C++, another may use only the Python wrappers, and a third a mix of both if dynamic loading into a Python program were used. This flexibility means you can use Python where simplicity is important, but C++ where better performance may be desirable.

The major classes in the OSE C++ class library involved in providing this functionality are the `OTC_SVBroker`, `OTC_SVRegistry` and `OTC_EVAgent` classes along with various event classes. In a distributed application the `OTC_Exchange` class comes into play along with the various classes used to implement the interprocess communications mechanism.

## Service Naming

When using the C++ class library, implementation of a service agent entails the use of a number of different classes together. In the Python interface this has all been brought together in the `Service` class. If you wish to create your own instance of a service agent, you need only derive a class from the `Service` class and then instantiate it.

The most important aspect of creating a service agent is the need to assign it a name. This name is what is used by other services to access your particular instance of a service agent. Having selected a name, it should be supplied to the `Service` base class at the point of initialisation. If you wished to call your service "`alarm-monitor`", the constructor of your class might look as follows.

```
class AlarmMonitor(netsvc.Service):
  def __init__(self,name="alarm-monitor"):
    netsvc.Service.__init__(self,name)
```

In general there is no restriction on what you can put in a service name. It is suggested though that you avoid any form of whitespace or non printable characters so as to make debugging easier.

In assigning a name to a service agent, there is nothing to stop you from having more than one service with the same name. Often the ability to have more than one service with the same name is useful, but in other situations it may be regarded as an error. As a policy on how to handle more than one service with the same name will be dependent on the actual application, implementation of any scheme to deal with it is left up to the user.

If you want to query what the service name is for an instance of a service agent, it can be queried using the "`serviceName()`" member function. If you need to know the unique identity of a service agent, it can be queried using the "`agentIdentity()`" member function. Even when two services share the same name, they will still have distinct agent identities. These as well as other details relating to a service agent can also be obtained from the object returned by the "`serviceBinding()`" member function.

Note that the `Service` class ultimately derives from the `Agent` class and as such all features of the event system are also accessible from a service agent. The `Service` class also builds on the same model used by the `Agent` class with respect to destruction of an object instance and the cleaning up of circular references. As such the `Service` class contains a derived implementation of the "`destroyReferences()`" member function found in the `Agent` class. Any derived service agent should use this function in the same way as defined for the `Agent` class.

## Service Audience

When you create a service, the existance of that service will be broadcast to all connected processes. If you wish to restrict visibility of a service to just the process the service is contained in, or a subset of the connected processes, a service audience can be defined.

To define the service audience, an extra argument needs to be supplied to the `Service` base class when it is initialised. By default the service audience is "*" to indicate that knowledge of the service should be broadcast as widely as possible. Setting the service audience to an empty string, will restrict visibility of the service to the local process.

```
class AlarmMonitor(netsvc.Service):
  def __init__(self,name="alarm-monitor",audience="*"):
    netsvc.Service.__init__(self,name,audience)
```

Other values can be supplied for the service audience and their meaning will depend upon how the interprocess communications links of the service agent framework are configured. This aspect of the service audience field will be discussed when support for distributed applications is covered.

Note that in setting the service audience, you are also restricting your service agent as far as what services it can subscribe to. If you set the service audience to that indicating the local process only, you will only be able to subscribe to services which exist in the local process. This is because services in remote processes will not know anything about you. If you need to be able to subscribe to services no matter where they are, you would generally be best leaving the service audience set to the default value.

## Anonymous Service

Although referred to as a service, a service agent can act in the role of either a client or server. That is, as a client it is a user of other services and would not expect to have subscriptions made against it or receive requests. In this situation the name assigned to the service is immaterial and it is valid to supply an empty service name. In fact, if you do not explicitly supply a service name when initialising the `Service` base class, it will default to an empty string.

```
class AnonymousService(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self)
```

In general it is still preferable to supply a non empty value for the service name. Doing so will mean that the service agent will appear as a separate entity within any debugging tools and although the application itself may not need to use that service agent in the role of a server, you might still include functionality which can be used from debugging tools so you know what the service agent is doing.

## Service Groups

When a service agent is created, the name of the service is notionally listed in a global group. In respect of this global group, unless you track the coming into existance of every single service agent, there is no way to make conclusions about a subset of services. Even if you do track the creation of every single service agent, the only way you might be able to distinguish a service agent as belonging to some group, is to introduce into the name of the service agent some form of artificial naming hierarchy.

Rather than rely on an artificial means of grouping service agents based on the service agent names, a separate concept of service groups is implemented. To add a service agent to a specific group, the "`joinGroup()`" member function can be called at any point after the `Service` base class has been initialised. That is, adding a service agent to a service group does not specifically have to been done in the constructor but can be done at a later time. To remove a service agent from a service group it has joined, the "`leaveGroup()`" member function can be called.

```
class EquipmentAgent(netsvc.Service):
  def __init__(self,name,audience="*"):
    netsvc.Service.__init__(self,name,audience)
    self.joinGroup("equipment-agents")
```

As with service names, it is recommended that you avoid using any form of whitespace or unprintable characters in service group names. The empty service group should also not be used to avoid confusion with the global group.

## Service Registry

The service registry is where information about available services is recorded. Each process in a distributed applicaton has its own service registry. The service registry in a process will list any services which are local to that process as well as any of which knowledge has been imported into the process from a remote process.

That each process has its own service registry means that the service agent framework can work quite happily within the context of a single process, as well as within the context of a distributed application. That is, when you only have a single process it isn't necessary for that process to be connected to a central server for the system to work. In this respect, each service registry acts as a peer to other service registries and not in a client/server mode.

A further consequence of this is that even when a process is part of a distributed application and the central message exchange process is terminated, any processes which were connected to it are not forced to restart themselves. In this scenario, any interested parties would be notified of the fact that remote services are no longer accessible and would take any action as appropiate. When the central message process is restarted, processes would automatically reconnect, with interested parties being notified that the remote services are once more accessible.

Any service agent may make queries against its local service registry and get back an immediate result which reflects the current state of the service registry. A service agent may also subscribe to the service registry or aspects of it and be notified in real time of changes made to the service registry. When subscribing to the service registry itself, a service agent would be notified of all available services, when those services join or leave groups and when those services are withdrawn.

Subscribing to the service registry as a whole is a useful debugging tool as it can produce an audit trail relating to the creation and deletion of services as well as group memberships. When used as a debugging tool as well as in other cases, it may not be appropriate that a service agent be created merely that the service registry can be queried. To this end, the member functions of the `Service` class relating to the service registry are also available through the `Monitor` class. In fact, the `Service` class derives from the `Monitor` class.

To setup a subscription against the service registry as a whole, the member function "`subscribeRegistry()`" is used. A subscription to the service registry can later be removed using the member function "`unsubscribeRegistry()`".

```
class RegistryMonitor(netsvc.Monitor):
  def __init__(self):
    netsvc.Monitor.__init__(self)
    self.subscribeRegistry(self.announce)
  def announce(self,binding,group,status):
    if group == None:
      # global group
      action = "WITHDRAWN"
      if status == netsvc.SERVICE_AVAILABLE:
        action = "AVAILABLE"
      name = binding.serviceName()
      identity = binding.agentIdentity()
      print "SERVICE-%s: %s (%s)" % (action,`name`,identity)
    else:
      # specific group
      action = "LEAVE"
      if status == netsvc.SERVICE_AVAILABLE:
        action = "JOIN"
      name = binding.serviceName()
      identity = binding.agentIdentity()
      print "%s-GROUP[%s]: %s (%s)" % \
          (action,`group`,`name`,identity)
```

When making queries or subscriptions against the service registry, details of a specific service are returned in the form of a service binding object. This is the same type of object returned by the "`serviceBinding()`" member function of a specific service agent. Where an operation needs to refer to a particular service it will be usually done in terms of this service binding object rather than the information it carries.

Member functions of a service binding object which may prove useful include "`serviceName()`", "`agentIdentity()`", "`serviceAudience()`", "`processAddress()`" and "`serviceLocation()`". Of these, "`serviceLocation()`" returns either "`SERVICE_LOCAL`" or "`SERVICE_REMOTE`", giving an indication if the service is located in the same process or a remote process. The "`processAddress()`" member function will return an internal address relating to the actual process the service is located in.

Although the shorthand "`agentIdentity()`" member function provides a more readable value, the "`serviceAddress()`" member function is also provided and returns the internal address used to identify the service. Note though that if in a distributed application an intermediary process along the route to the actual service is restarted, when all processes reconnect, the service address will be different where as the process identity and agent identity would be the same. This reflects the fact that it is still the same service, but the route used to contact the service has now changed as the intemediary process was restarted.

When subscribing to the service registry as a whole, each notification will also include a group and status value. When the group is "`None`", the notification refers to either the availability or withdrawal of a service. For any other value of group, it indicates that a specific service is joining or leaving that group. Whether a service has become available or has been withdrawn, or similarly whether a service has joined or left a group is given by the status value. When the status is "`SERVICE_AVAILABLE`", a service has become available or has joined a group as appropriate. When the status value is "`SERVICE_WITHDRAWN`" the service has either been withdrawn or has left a group as appropriate.

Note that when the status indicates that a service has become available it doesn't mean that the service only just got created. In the case that a service is in a remote process, it may be the case that a service has existed for some time, but because the local process has only just connected into a distributed application it has only just become aware of that fact.

Similarly, when a service is withdrawn, if the service was in a remote process it means the service can no longer be contacted. This may have occurred because the service itself has been destroyed, the process in which the service existed has been destroyed or that an intemediary process involved in the communication path for contacting that process has been destroyed and the remote process is currently no longer contactable.

By subscribing to the service registry it is possible to receive in real time notitifications regarding the availability of services as such events happen. If you only wish to find out which services are available at a particular instant in time, you can use the "`serviceAgents()`" member function. Note that depending on the number of service agents available, calling this member function repetitively can incur significant overhead. If possible this member function should be used sparingly and a subscription against the service registry used instead.

## Service Announcements

If a service agent subscribes to the registry using a specific service name, the service agent will be notified when any service with that name becomes available or is subsequently withdrawn. When subscribing to the registry using a specific service name, no notification is given regarding groups that those same services may join.

```
class ServiceMonitor(netsvc.Monitor):
  def __init__(self,name):
    netsvc.Monitor.__init__(self)
    self.subscribeServiceName(self.announce,name)
  def announce(self,binding,status):
    action = "WITHDRAWN"
    if status == netsvc.SERVICE_AVAILABLE:
      action = "AVAILABLE"
    name = binding.serviceName()
    identity = binding.agentIdentity()
    print "SERVICE-%s: %s (%s)" % (action,`name`,identity)
```

The name of the member function for subscribing to the existance of a service agent by name is "`subscribeServiceName()`". A subscription can be cancelled by calling the member function "`unsubscribeServiceName()`".

Having identified a particular service agent, it is often useful to know when that specific service agent is no longer available. The notifications provided when you call the member function "`subscribeServiceName()`" will tell you that, but if the service binding had been received through some other means and you weren't receiving the notifications, it is preferable that you be able to receive a notification just in relation to the specific service agent you are using. In this case, the service address can be obtained from the service binding by calling "`serviceAddress()`" and the member function "`subscribeServiceAddress()`" used instead. This subscription can be cancelled by calling the "`unsubscribeServiceAddress()`" member function.

```
class ClientService(netsvc.Service):
  def __init__(self,binding):
    netsvc.Service.__init__(self)
    address = binding.serviceAddress()
    self.subscribeServiceAddress(self.announce,address)
    self._binding = binding
    # start using service
  def self.announce(self,binding,group,status):
    if group == None:
      if binding.agentIdentity() == self._binding.agentIdentity():
        if status == netsvc.SERVICE_WITHDRAWN:
          self.unsubscribeServiceAddress(binding.serviceAddress())
          self._binding = None
          # stop using service
```

## Group Announcements

If a service agent subscribes to the service registry using a specific service group, it will be notified when any service joins or leaves that group. Notice that a service has left a particular group will also be be notified when the service is withdrawn and the service hadn't explicitly left the group before hand. The member functions relating to service group subscriptions are "`subscribeService-Group()`" and "`unsubscribeServiceGroup()`".

Subscription to a service group is most often used as a way of finding out what services exist which perform a certain function. As an example, service agents which provide an interface to equipment in a telecommunications network could join a particular group. A service which has the task of monitoring alarms generated by the same equipment could then subscribe to that service group and be notified about each equipment agent. Knowing about each equipment agent, the alarm monitor could then subscribe to any alarm reports generated by the equipment agents.

```
class EquipmentMonitor(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self,"equipment-monitor")
    self.subscribeServiceGroup(self.announce,"equipment-agents")
  def announce(self,binding,group,status):
    if status == netsvc.SERVICE_AVAILABLE:
      self.monitorReports(self.alarm,binding,"alarm.*")
    else:
      self.ignoreReports(binding)
  def alarm(self,service,subject,content):
    print subject,content
```

By using a service group it is therefore possible to make an application respond dynamically to the introduction of new service agents. In the case of the equipment alarm monitor for a telecommunications network, it would not be necessary to hardwire in details of the equipment. Instead, when adding a new piece of a equipment, the service agent providing an interface to that equipment need only add itself to the appropriate service group.

Such a mechanism could also be used to monitor alarms raised as a result of problems in the application itself and need not be alarms generated by some piece of equipment. This mechanism could therefore also be used as the basis of an application health monitoring system.

## Service Lookup

The ability to subscribe to the service registry provides a means of tracking the existance of service agents over time. The alternative to subscribing to the service registry to find out about available services, is to do a lookup against the service registry. Performing a lookup will tell you immediately what service agents exist at that particular point in time. No subscription will be registered when doing a lookup though, so if you need to know when a service agent is subsequently withdrawn it still may be appropriate to subscribe to the registry using the address of the specific service agent you use.

```
class PollingService(netsvc.Service):
  def __init__(self,name,period=60):
    netsvc.Service.__init__(self)
    self._name = name
    self._period = period
    self.initiateRequests("poll")
  def initiateRequests(self,tag):
    bindings = self.lookupServiceName(self._name)
    for binding in bindings:
      service = self.serviceEndPoint(binding)
      # presume remote service provides uptime method
      id = service.uptime()
      self.processResponse(self.handleResult,id)
    self.startTimer(self.initiateRequests,self._period,"poll")
  def handleResult(self,result):
    address = self.currentResponse().sender()
    binding = self.lookupServiceAddress(address)
    if binding != None:
      print binding.agentIdentity(),result
```

A number of different types of lookup can be made against the service registry. The first two allow you to lookup all service agents which have a particular service name, or all service agents which are currently a member of a specific service group. The two member functions corresponding to these lookups are "`lookupServiceName()`" and "`lookupServiceGroup()`". Both these lookup functions return a list of service binding objects corresponding to the service agents found. If there are no service agents matching the search criteria, an empty list is returned.

The third type of lookup is that of looking up a specific service agent using its service address. In this case you will need to have been able to obtain the service address by some other means first. The member function here is "`lookupServiceAddress()`". The result will be the service binding object corresponding to that service agent, or `None` if the service agent is no longer available.

In order to obtain a list of all services known of by the service registry, the member function "`serviceAgents()`" can be used. This should however be used sparingly because of the overhead which might be incurred when there are large numbers of services. If possible, subscription against the service registry should still be used if it is necessary to track all available services. Overhead can be reduced by using subscription and caching the results as Python data structures, with Python objects accessing the cache directly. This avoids the translation from C++ data structures to Python data structures.

Similarly to service agents, a list of all service groups can be obtained by calling the member function "`serviceGroups()`". If it is necessary to determine which service groups a particular service agent is a member of, an optional argument can be supplied to "`serviceGroups()`", that argument being the service address of the service agent of interest.

# Service Reports

When using the service agent framework, in addition to being able to subscribe to the service registry in order to receive announcements regarding the existence of services, it is also possible to subscribe to actual services. When subscribed to a service, if that service publishes a report with a subject matching the subscription, the subscriber will automatically receive it.

Referred to as publish/subscribe, this is a common feature of packages implementing message oriented middleware services. Note that in this implementation, the design and interface are driven by simplicity. As a result, the implementation is not underlaid by persistent message queues. While a subscriber exists and is known of by the publisher, it will receive any reports for which it has a valid subscription. If a subscriber is destroyed but is subsequently restarted, it will only receive reports published from that time onwards, it will not receive reports which may have been published in the time that it was offline.

The system design can therefore be likened to a system implementing instant messaging as opposed to a mailing list. In instant messaging you will only see those messages which are posted into a group while you are online, whereas with a mailing list any messages posted while you were away will still be there for you to see when you log back in.

As a basic system, this model of operation is suitable for many applications, but not all. If you are developing a system where it is imperative that you never miss a message, you would be advised to purchase a commercial message oriented middleware package. You will of course have to deal with the extra complexity and cost that entails.

## Publishing Reports

If a service agent needs to publish a report, the member function "`publishReport()`" is used. In publishing a report, it will generally be the case that a service agent does it without caring who may actually be subscribed to that report. This is often referred to as anonymous publishing and results in a more loosely coupled system which can adjust dynamically to changes. That is, it is not necessary to hardwire into a service to whom it should send a report, instead, a service which is interested in the report will subscribe to it and the underlying system will handle everything else.

```
self.publishReport("subject.string","value")
self.publishReport("subject.integer",12345)
self.publishReport("subject.float",1.2345)
self.publishReport("subject.list",[1,2,3,4,5])
self.publishReport("subject.dict",{"one":1,"two":2})
```

When publishing a report, a service agent needs to supply a subject which in some way identifies the purpose of the report, as well as the content of the report. It is through subscription to specific subjects that subscribers will indicate their interest in specific reports. The subject name assigned to a report can have any value, but it is suggested that a hierachical naming convention be used. That is, use one or more name components, where each component is separated by a period.

```
heartbeat
news.local.sanitation
news.domestic.politics
notifications.shutdown
```

By using a naming hierarchy, it becomes possible to aggregate reports into groupings which can then be easily subscribed to as a whole. Note that there is nothing special about a period as the separator for the name components. Other separators which are often used for performing the same task are a slash or a colon.

## Monitoring Reports

A desire to subscribe to reports published by another service is indicated by a service agent calling the "`monitorReports()`" member function. In setting up such a subscription, the service agent must supply a callback function to be called when a report is received, the name of the service or the service binding object of the specific service agent to which it is subscribing and an indication of what reports it is interested in.

In the simplest case, a subscription can supply the exact same subject name under which a report is published. Alternatively, it can use special wildcard characters to allow it to pick up reports published against related subjects. The two special wildcard characters which can be used are "`*`" and "`?`". These can be incorporated anywhere in the subscription pattern.

The "`?`" can be used to match a single character within the subject name, where as a "`*`" will match any number of characters. Note that each will match any character, including a period or slash. As such,

a subscription of "`system.*`" will match "`system.time`" and "`system.statistics.users`", but not "`system`". To subscribe to any reports from a particular publisher, "`*`" would be used.

Note that the subscription pattern described here is the default. It is actually possible within the C++ implementation of a service agent to override the default and supply an alternate matching algorithm. For example, in a bridge to the TIB/Rendevous package, a service agent would most likely redefine the matching algorithm to match that of that package. Therefore, when subscribing to a service agent, always check first exactly which scheme it uses.

```
class Publisher(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__("publisher")
    self.joinGroup("publishers")
    self.publishReport("system.ctime",netsvc.DateTime(),-1)
    self.startTimer(self.timeout,10,"heartbeat")
  def timeout(self,tag):
    self.publishReport("system.time",netsvc.DateTime())
    self.startTimer(self.timeout,10,"heartbeat")

class Subscriber(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self)
    # subscribe to any service agent with name "publisher"
    self.monitorReports(self.report,"publisher","system.*")
  def report(self,service,subject,content):
    binding = self.currentReport().publisher()
    identity = binding.agentIdentity()
    publisher = "(%s/%s)" % (‘service‘,identity)
    if subject == "system.ctime":
      now = str(netsvc.DateTime())
      print "%s became available at %s" % (publisher,now)
      print "%s originally started at %s" % (publisher,str(content))
    elif subject == "system.time":
      print "%s was still alive at %s" % (publisher,str(content))
```

When called, the callback supplied by the subscriber will be passed three arguments. These are the service binding object for the service agent which published the report, the subject under which the report was published and the content of the report.

The service binding object for the service agent which published the report is provided for a number of reasons. The first is that since more than one service agent may use the same service name, it is possible that a subscription based on service name might result in responses from more than once service agent. The service binding object is therefore supplied so that it is possible to distinguish from whom a report originated. The service binding object may also be used to identity a particular service agent and send a request to it. This may be less of an issue if when subscribing to a service agent, the service binding object for the specific service agent of interest is used as opposed to a service name. This eliminates the possibility of getting reports from unrelated service agents using the same service name.

```
class Subscriber(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self)
    # subscribe to the service group "publishers"
    self.subscribeServiceGroup(self.announce,"publishers")
  def announce(self,binding,group,status):
    if status == netsvc.SERVICE_AVAILABLE:
      # now subscribe to service agent which is member of group
      self.monitorReports(self.report,binding,"system.*")
    else:
      self.ignoreReports(binding)
  def report(self,service,subject,content):
    binding = self.currentReport().publisher()
    identity = binding.agentIdentity()
    publisher = "(%s/%s)" % (‘service‘,identity)
    if subject == "system.ctime":
      now = str(netsvc.DateTime())
      print "%s became available at %s" % (publisher,now)
      print "%s originally started at %s" % (publisher,str(content))
    elif subject == "system.time":
      print "%s was still alive at %s" % (publisher,str(content))
```

As expected, the subject is that under which any report was published. As to the content of the report, this is not limited to being a string, but can be any of the basic Python scalar types, a list, tuple or dictionary, as well as the `None` type and a number of extended types. User defined scalar types can also be used providing that appropriate encoders/decoders are available.

If you wish to cancel a subscription to a service, the "`ignoreReports()`" member function should be used. This should be supplied the name of the service and the exact same subject pattern used when subscribing to the reports in the first place. If no subject pattern is supplied, all subscriptions against that service name will be removed.

## Lifetime of Reports

When publishing a report, the report will be sent to any service agents which have a current subscription which matches the subject associated with the report. The default behaviour is then such that the publishing service forgets all about the report. In this case, if a new subscription arrived immediately after, it would only be sent any reports which were published after its subscription was received. The new subscriber would not receive a copy of the report which was published just before its subscription was received.

In some situations however, it is desirable that a new subscriber be able to obtain the last report which may have been published against any subject it is interested in. This is useful in the context that a report is used to reflect the status of a service. By being able to obtain the last published report, a subscriber can know the current state of the service immediately and doesn’t have to explicitly request it or wait for the status to change.

For such cases, it is possible to supply an optional lifetime for a report. That is, a time in seconds for which the report should be cached by the publishing service. When such a value is supplied, if a subscription arrives within that time, it will be sent a copy of that report. If a value of "`-1`" is supplied for the lifetime, it will effectively cache the report indefinitely.

```
# publish and cache indefinitely
self.publishReport("system.status","idle",-1)

# publish and cache for 60 seconds
self.publishReport("system.action","twiddle thumb",60)

# publish but don't cache
self.publishReport("system.thought","bored")
```

A cached report will only be discarded if a new report is published against the same subject, or the lifetime specified expires. If a new report published against the same subject has no lifetime associated with it, the cached report will be discarded, but the new report will not be cached. Note that with this mechanism, only the last report published on a specific subject will ever be cached when a lifetime value is provided.

To make the implementation as simple as possible, a report which has been cached against a subject with a finite lifetime and which has expired, will only be discarded when a new report with the same subject name is published, or a new subscription arrives which would have matched the subject. This is done to avoid having to setup internal timers to trigger destruction of the report at the moment it expired.

A consequence of this approach however, is that a report may consume resources unnecessarily beyond the lifetime which it was supposed to exist. If this becomes an issue, it is possible for a service agent to periodically purge any expired reports itself. This can be done by calling the member function "`purgeReports()`".

```
class Publisher(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self,"publisher")
    # purge expired reports every 15 minutes
    self.scheduleAction(self.purgeReports,"*/15 * * * *")
```

In addition to being able to explicitly purge expired reports for performance reasons, a service agent may also prematurely expire and purge reports which are older than a certain time. The member function for this is "`expireReports()`" and accepts a subject pattern and optional age in seconds. The age defaults to "`0`" which would result in any cached report matching the subject pattern being immediately expired and purged. If a non zero value for age is supplied, only reports which were older than that age would be expired and purged. To apply this to all cached reports, regardless of subject, the "`expireAllReports()`" member function can be used.

Although "purgeReports()" exists specifically to deal with potential performance issues in a very limited number of cases, the "expireReports()" and "expireAllReports()" member functions are useful where a service may have reset itself and it was necessary to discard all cached reports so that new subscribers didn't receive them.

## Identity of Subscribers

In most circumstances the identity of a subscriber is not important, however, such information can be quite useful in a few circumstances. At present this information is available by overriding a method in the service agent base class.

```
class Publisher(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self,"publisher")
  def handleSubscription(self,subscription):
    subscriber = subscription.subscriber()
    if subscription.status() == netsvc.SUBSCRIPTION_REQUESTED:
      if self.matchSubject(subscription.subject(),"system.time"):
        self.sendReport(subscriber,"system.time",netsvc.DateTime())
```

One can use this feature in preference to caching reports when they are published. That is, rather than caching a report when it is published so that a new subscriber automatically receives it, generate the report only when the subscription arrives. Obviously however, in this approach we would only want to have the report sent to the particular subscriber and not to all subscribers as they would potentially get duplicates otherwise.

To cater for this scenario, the member function "sendReport()" is supplied. In this variant of report publishing, the first argument is the service binding object of the subscriber obtained from the subscription notification. This report will only be sent to the subscriber in question and will not be cached. Note that if a report was also cached against the subject in question, the subscriber would still receive it as well. Both anonymous publishing and targeted reports should therefore not be used in combination for a specific subject as it may give undesired results.

The member function "matchSubject()" is supplied to assist in determining if the subject pattern contained in the subscription matches that of a particular subject. The first argument to "matchSubject()" should be the pattern and the second the actual subject. Although not used here, the opposite to the status value "SUBSCRIPTION_REQUESTED" is the value "SUBSCRIPTION_WITHDRAWN".

Note that if the "sendReport()" member function is used to send a report and the recipient has a subscription against the publishing service, but doesn't have a subscription against that service matching that subject, the report will not be delivered via the callback it originally supplied with its subscription. A similar situation is where a service receives an unsolicited report, or had since unsubscribed from the reports. In these cases there is no current callback in place for reception of the report. When this occurs the member function "unexpectedReport()" will be called. A service agent may if it desires override this member function so as to deal with any such unexpected reports.

A further use of the mechanism for identifying a subscribers identity, is so that subscriptions can be tracked and for processing or interception of data only to be undertaken while there are subscribers interested in the results. This avoids unnecessarily publishing reports when it is known there would be no one to send them to.

```
class LogMonitor(netsvc.Service):
  def __init__(self):
    name = "logmon@%s" % netsvc.processIdentity()
    netsvc.Service.__init__(self,name)
    self._logger = netsvc.Logger()
    self._channels = {}
  def notify(self,channel,level,message):
    agent = channel[1:-1]
    report = {}
    report["agent"] = agent
    report["level"] = level
    report["message"] = message
    self.publishReport(agent,report)
  def handleSubscription(self,subscription):
    agent = subscription.subject()
    channel = "(%s)" % agent
    if subscription.status() == netsvc.SUBSCRIPTION_REQUESTED:
      if len(agent) != 0:
        subscriber = subscription.subscriber().agentIdentity()
        if not self._channels.has_key(channel):
          self._channels[channel] = []
          self._logger.monitorChannel(channel,self.notify)
          self._channels[channel].append(subscriber)
    else:
      if self._channels.has_key(channel):
        subscriber = subscription.subscriber().agentIdentity()
        if subscriber in self._channels[channel]:
          index = self._channels[channel].index(subscriber)
          del self._channels[channel][index]
          if len(self._channels[channel]) == 0:
            del self._channels[channel]
            self._logger.monitorChannel(channel,None)

logger = netsvc.Logger()

class Publisher(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self,"publisher")
    self._channel = "(%s)" % self.agentIdentity()
  def debug(self,message):
    logger.notifyChannel(self._channel,netsvc.LOG_DEBUG,message)
```

In this use of subscription information, the subscription to a specific subject is used to trigger interception of messages logged via the logger interface. For the time that subscriptions exist for a particular

subject corresponding to a log channel, the log messages on that log channel will be intercepted and published. This can be useful as a remote debugging mechanism and will not unnecessarily load the process as information is only being captured and published when it is actually required.

## Existence of Publishers

When a subscription to a service is made, if the service holds any cached reports with a subject matching the subscription, the subscriber will receive them immediately. If however there were no such reports, the subscriber will not receive any reports until some are published having a subject which matched its subscription. Even when there are reports which can be sent back immediately, if there are reports against multiple subjects, there is no guarantee as to which order they will be received in.

As a consequence, using the reception of a report as an indicator that a service has become available is not a good approach to take. This is because a report may not be received until some time after the service became available and the subscription accepted. Further, there is no indication when the service is no longer available.

One way as previously described of knowing when a service becomes available or when it is withdrawn, is to subscribe to the service registry. Although this will work, if you have restricted the service audience of your service agent, it will also possibly tell you about services outside of the scope of what you can subscribe to.

To avoid this difficulty, the member function "`handlePublisherNotification()`" is provided. This member function can be overridden in your service agent and will be called only when a subscription has been matched up and accepted by the service being subscribed to. Note that this notification will only occur for the first subscription against a particular service agent.

This member function will also be called to acknowledge withdrawal of the last subscription against a particular service agent, or when a service agent to which you were subscribed has been withdrawn.

```
class Subscriber(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self)
    self.monitorReports("publisher","*")
  def handlePublisherNotification(self,notification):
    name = notification.publisher().serviceName()
    identity = notification.publisher().agentIdentity()
    publisher = "(%s/%s)" % (‘name‘,identity)
    if notification.status() == SERVICE_AVAILABLE:
      print "%s AVAILABLE" % publisher
    else:
      print "%s WITHDRAWN" % publisher
```

Knowledge of when a subscription has been accepted or when the service agent subscribed to has been withdrawn can be useful when there is more than once service agent with the same name, and it is nec-

essary to track the lifetime of each. It is also useful where it might be necessary to immediately send off a request to each service agent to obtain information not available via published reports.

# Service Requests

The ability within the service agent framework to find out what services exist and the ability of a service agent to publish reports can be useful in itself, but more often than not one wants to make a specific request of a service to perform some action. In most cases such an action would result in a response, be it the return of data related to the request being made or an error indication. Referred to as request/reply, this is probably the most fundamental feature of message oriented middleware software.

As with the implementation of the publish/subscribe feature, simplicity has been a major driving force in influencing the design. As a result, the implementation of the request/reply feature should not be likened to that of point to point messages using persistant messages queues. In this implementation, if a service to which you want to send a request doesn't exist you will not be able to send your request, nor is a service able to receive any requests sent when it wasn't running.

Although persistant message queues are not a feature of this implementation, the request/reply and publish/subscribe features can actually be seen as sitting at a lower level of abstraction. As a result, it would be possible to build on top of these features and implement persistent message queues and gauranteed modes of delivery if required. For many systems such features aren't required though, so they are not implemented with the aim being to make the software as simple as possible to use and understand.

## Sending a Request

In order to send a request to a service you need to first obtain a service endpoint object. This is a special Python object which holds an internal reference to the service binding object for the service and which will automatically dispatch your request for you. To obtain a service endpoint, the member function "`serviceEndPoint()`" is used.

When invoking "`serviceEndPoint()`", the member function needs to be supplied with either a service binding object for the particular service agent to which you wish to send the request, or the name of the service. When a service name is supplied, a lookup will be made against the service registry and the first service agent found with that service name will be used.

To invoke the request against the remote service agent, the service endpoint object is used as a proxy. That is, a member function call is made against the object as if it were the actual service object you wished to call. The only difference is that the call isn't synchronous but asynchronous. This means that the result is not returned immediately.

As to the parameters to the call, multiple arguments can be supplied, with any of the basic Python scalar types, a list, tuple, dictionary, the `None` type, as well as a number of extended types being able to be used. User defined scalar types can also be used providing that appropriate encoders are available. Note that keyword arguments cannot be used and will be ignored.

```
class PagerClient(netsvc.Service):
  def __init__(self,number,message):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      service.send(number,message)
```

When a service endpoint is created by using a service name, you should always check whether a service agent with that name could actually be found. This is done by doing a truth test against the service endpoint object or comparing it to `None`. Note that although it may equate to `None`, the service endpoint object is a distinct object in its own right. If you don't check the validity of the service endpoint object and still make a request against the service, a special exception indicating that such a service isn't available will be raised.

```
class PagerClient(netsvc.Service):
  def __init__(self,number,message):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    try:
      service.send(number,message)
    except netsvc.ServiceUnavailable:
      # ...
```

Obviously the service name by itself can only be used if you don't care which instance of a service is used when there is more than one. If you wanted to select a specific service agent, or wanted to be able to send a request to all service agents with the same service name, you would need to perform a lookup against the service registry to obtain the full list of service agents.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    bindings = self.lookupServiceName(name)
```

```
    for binding in bindings:
      service = self.serviceEndPoint(binding)
      if service:
        service.reset()
```

## Handling a Response

When you send a service request, you do not get an immediate response back. That is, the call is asynchronous. If you want to be able to capture any response generated from a request, you need to capture the conversation id associated with the request and then register a callback to handle the response. The conversation id is the value returned when you make the call against the service endpoint object. Having obtained the conversation id you must then register a callback to handle the response using the member function "`processResponse()`". If you also want to be notified that the request has failed, you will also need to set up a separate callback using the "`processFailure()`" member function.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processReponse(self.uptimeResponse,id)
      self.processFailure(self.uptimeFailure,id)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self):
    print "failure"
```

The callbacks which you put in place to handle the result and/or failure will be automatically deregistered when a response is received. This will be the case whether the response is valid or was a failure indication. Prior to having received a response, if you decide you are no longer interested in the response, you can call the member function "`ignoreResponse()`" supplying the conversation id. If you are submitting multiple requests in one go, you must call the "`processResponse()`" and/or "`processFailure()`" member functions for a conversation id before you send any subsequent request.

Note that prior to the release of OSE 7.0pl5, instead of using the "`processResponse()`" and "`processFailure()`" member functions, one would use the "`monitorResponse()`" method. This method in effect combined the operation of both of the new methods albeit it with subtle differences as far as the arguments the callback would be passed and the functionality it implemented. Using the new methods it is possible to register separate callbacks for handling of the result versus a failure. It is even possible to only register interest in one or the other of the result or a failure notification. The "`monitorResponse()`" member function should as a result now be viewed as deprecated and should not be used.

## Identifying a Response

If a callback is being registered to handle the response from multiple service requests, you will most likely need to be able to identify to which request a response belongs to. To get the conversation ID of the original request, the "`conversationId()`" member function can be called.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    bindings = self.lookupServiceName(name)
    for binding in bindings:
      service = self.serviceEndPoint(binding)
      if service:
        id = service.uptime()
        self.processResponse(self.uptimeResponse,id)
        print "request",binding.agentIdentity(),id
  def uptimeResponse(self,result):
    id = self.conversationId()
    print "result",id,result
```

Instead of requesting the conversation id, it is also possible to define your callback so as to take two arguments instead of one, these being the conversation id and the result instead of just the result.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    bindings = self.lookupServiceName(name)
    for binding in bindings:
      service = self.serviceEndPoint(binding)
      if service:
        id = service.uptime()
        self.processResponse(self.uptimeResponse,id)
        print "request",binding.agentIdentity(),id
  def uptimeResponse(self,id,result):
    print "result",id,result
```

These are not keyword arguments, but positional parameters which the code which calls the callback function supplies or not based on the number of arguments the callback accepts. In other words, the callback must accept the appropriate number of arguments as necessary and in the specified order. If you know that the remote method being called doesn’t actually return a valid response, ie., it returns a void or null response, you can even leave out the parameters altogether.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    bindings = self.lookupServiceName(name)
    for binding in bindings:
      service = self.serviceEndPoint(binding)
      if service:
```

```
        id = service.reset()
        self.processResponse(self.resetResponse,id)
        print "request",binding.agentIdentity(),id
    def resetResponse(self):
      print "result"
```

In addition to "`conversationId()`" the member function "`currentResponse()`" is also available. This member function returns an object providing both the "`conversationId()`" and "`sender()`" member functions. If you need the service binding object for the service agent who sent the response, you can perform a lookup against the service registry using the service address provided by "`sender()`". Note though that you shouldn't assume that the service binding object will be available as the remote service may have been withdrawn by the time you make your query.

## Detecting a Failure

If you send a service request to a service agent and you need to detect if a failure occurs, you will need to have registered a callback using the "`processFailure()`" member function. A failure may occur due to the service not supplying a method to handle the request you made, an incorrect number of arguments being supplied, an error within the method being called or because the remote service agent was withdrawn before a response was received.

When a failure does occur, the details of the failure can be obtained in a number of ways. If the callback you provide doesn't take any arguments, you can obtain a failure object detailing the error which occurred by calling the "`currentFailure()`" member function. The member functions provided by the failure object are "`error()`", "`description()`", "`origin()`" and "`details()`". The conversation id associated with the request which failed can be obtained using the member function "`conversationId()`".

Of the member functions provided by the failure object, the "`error()`" member function returns an integer error code. The "`description()`" member function returns a text description of the error. The "`origin()`" member function returns a string which in some way identifies the origin of the error and "`detail()`" may contain as text, extra details relating to the error which has occurred.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processResponse(self.uptimeResponse,id)
      self.processFailure(self.uptimeFailure,id)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self):
    id = self.conversationId()
    failure = self.currentFailure()
```

```
    if failure.origin() == "netsvc" and \
        failure.error() == netsvc.SERVER_METHOD_UNAVAILABLE:
      # method didn't exist
```

As an alternative to using the "`conversationId()`" member function to obtain the conversation id of the failed request, if the callback accepts a single argument, the conversation id will be passed as an argument to the callback function.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processResponse(self.uptimeResponse,id)
      self.processFailure(self.uptimeFailure,id)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self,id):
    failure = self.currentFailure()
    if failure.origin() == "netsvc" and \
        failure.error() == netsvc.SERVER_METHOD_UNAVAILABLE:
      # method didn't exist
```

This ability to have details of the failure supplied as arguments to the callback function also extends to the contents of the failure object if the callback function accepts an additional four parameters in addition to that for the conversation id.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processResponse(self.uptimeResponse,id)
      self.processFailure(self.uptimeFailure,id)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self,id,error,description,origin,details):
    if origin == "netsvc" and error == netsvc.SERVER_METHOD_UNAVAILABLE:
      # method didn't exist
```

These are not keyword arguments, but positional parameters which the code which calls the callback function supplies or not based on the number of arguments the callback accepts. In other words, the callback must accept the appropriate number of arguments as necessary and in the specified order.

## Lack of Response

When you send a request, there is no gaurantee that the remote service agent hasn't been destroyed even before it receives your request. If a remote service agent delays sending an immediate response to your request, the problem might also arise that the remote service agent is destroyed before it completes the response. Finally, an intermediate process relaying your request might be shutdown or crash meaning either the request or response is lost.

In order to handle these situations, when the "`processFailure()`" member function is used to register interest in the failure of a request, it will automatically setup a subscription on the existance of the remote service agent against which the request has been made. In the event that the remote service agent becomes unavailable before a response is received, an application error will be returned as a failure to provide notification of this occuring.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processResponse(self.uptimeResponse,id)
      self.processFailure(self.uptimeFailure,id)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self,id,error,description,origin,details):
    if origin == "netsvc" and error == netsvc.SERVER_APPLICATION_ERROR:
      # request has failed, possibly because no response was received
```

Note that the "`monitorResponse()`" member function which has been made deprecated as of OSE 7.0pl5, does not setup a subscription to the existance of the remote service agent. Thus, if you are using this older member function to catch the failure of a request, you will not get any failure notification in these circumstances.

Although the "`processFailure()`" member function will ensure that a failure is returned if no response is received prior to the remote service agent becoming unavailable, programming errors or external communications failures in code associated with the remote service agent might still result in no response being received where the remote service agent still exists. If this is an issue and you also want to implement a timeout whereby if no response has been received within a certain period of time, a timeout value can be supplied when you call the "`processFailure()`" member function.

```
class Client(netsvc.Service):
  def __init__(self,name):
    netsvc.Service.__init__(self,"","")
    service = self.serviceEndPoint("SMS")
    if service:
      id = service.uptime()
      self.processResponse(self.uptimeResponse,id)
```

```
      self.processFailure(self.uptimeFailure,id,60)
  def uptimeResponse(self,result):
    print result
  def uptimeFailure(self,id,error,description,origin,details):
    if origin == "netsvc" and error == netsvc.CLIENT_REQUEST_TIMEOUT:
      # timeout occurred
```

When a timeout occurs, it will be notified as a request failure. The timeout should be the maximum number of seconds to wait. The callback will be automatically deregistered and if the response did subsequently arrive it would be ignored. If you wanted a timeout to occur but didn't want the callback to be deregistered, you would need to create your own timer. If that timer uses the conversation id corresponding to the request as the timer name, the timer will be automatically stopped if a response does actually arrive. You should not use the conversation id to set up a timer if you have already defined a timeout when calling the member function "`processFailure()`" as internally it will use the conversation id for its own timer.

## Servicing a Request

When you send a request, if the remote service agent is implemented using the Python interface, not just any member function of the service can be called. In order that a member function of a service can be called, the service agent must have exported it as a public method. This is done by calling the member function "`exportMethod()`" and it would normally be done from within the constructor of the service agent.

```
class PagingService(netsvc.Service):
  def __init__(self,name="SMS"):
    netsvc.Service.__init__(self,name)
    self.exportMethod(self.time)
    self.exportMethod(self.uptime)
    self.exportMethod(self.send)
  def time(self):
    return netsvc.DateTime()
  def uptime(self):
    # ...
  def send(self,number,message):
    # ...
```

By default the method name associated with the member function will be its actual name. If you wish to export a member function under a different method name, the method name can be supplied as an extra argument to the "`exportMethod()`" member function.

```
class PagingService(netsvc.Service):
  def __init__(self,name="SMS"):
    netsvc.Service.__init__(self,name)
    self.exportMethod(self.sendMessage,"send")
  def sendMessage(self,number,message):
    # ...
```

The reason for requiring that methods be explicitly exported is that it would usually be quite dangerous to allow open access to all member functions of a class. This is because any class is likely to implement methods to assist in intermediate stages of processing a request. Providing automatic access to such member functions could compromise the operation of the class.

When a method is invoked as a result of a service request, the default behaviour will be that the value returned from the method will be what is returned to the caller as the response. If necessary, a method may explicitly indicate that a failure response should instead be returned. A method can also indicate that a delayed response will be sent. This latter case is useful when the service needs to do something first in order to generate a suitable response.

## Generating a Failure

If a method encounters an error and raises an exception this will be caught by the service agent framework and a failure response will be generated. The value of the origin for this type of failure will be "`netsvc`" and the value of the error code will be "`SERVER_APPLICATION_ERROR`". If you want to generate a failure response which is specific to your application, you should catch any exceptions and indicate the type of failure response by calling the member function "`abortResponse()`".

```
class Database(netsvc.Service):
  def __init__(self,name="database",**kw):
    netsvc.Service.__init__(self,name)
    self._database = MySQLdb.connect(**kw)
    self.exportMethod(self.execute)
  def execute(self,query,args=None):
    try:
      cursor = self._database.cursor()
      cursor.execute(query,args)
      if cursor.description == None:
        return cursor.rowcount
      return cursor.fetchall()
    except MySQLdb.ProgrammingError,exception:
      details = netsvc.exceptionDetails()
      self.abortResponse(1,"Programming Error","db",details)
    except MySQLdb.Error,(error,description):
      self.abortResponse(error,description,"mysql")
```

The four arguments to the member function "`abortResponse()`" are the error code, the description of the error, the origin and any additional details. It is recommended that an origin which clearly identifies the source of the error, or namespace from which the error codes are derived always be used. If an origin is not used, it becomes impossible to programmatically deal with an error when different aspects of a service generate overlapping error code sets.

Note that the "`abortResponse()`" member function will in turn raise its own special exception. When this exception is caught by the service agent framework, it will be translated into a failure response as described by the arguments used to call "`abortResponse()`". As a new exception is

raised, you should avoid an except clause which catches all exceptions in any code which encloses code which might call "`abortResponse()`". Alternatively, you should explicitly pass on exceptions of type `ServiceFailure`.

```
try:
  self.execute(...)
except netsvc.ServiceFailure:
  raise
except:
  self.abortResponse(...)
```

If many of the public methods of a service generate the same type of exceptions, rather than provide code to catch the exceptions in every method, it is possible to override the member function "`executeMethod()`". This member function is called by the service agent framework to call the actual member function referred to by a service request. It is important to preserve the existing functionality of this method otherwise service requests will not execute correctly.

```
class Database(netsvc.Service):
  # ...
  def executeMethod(self,name,method,params):
    try:
      return netsvc.Service.executeMethod(self,name,method,params)
    except MySQLdb.ProgrammingError,exception:
      details = netsvc.exceptionDetails()
      self.abortResponse(1,"Programming Error","db",details)
    except MySQLdb.Error,(error,description):
      self.abortResponse(error,description,"mysql")
```

The member function "`executeMethod()`" might also be overridden if you want to track what requests are being made against a service. The arguments to the member function are the name of the method, the actual member function and the parameters to be supplied when the member function is called.

## Delaying a Response

In a distributed application, it is sometimes the case that when a method is called it doesn't have the information necessary to generate an immediate response. This may be the case where it needs to initiate its own service requests to accumulate the data needed to generate the result. Because the service agent framework is based on an event driven system, it is not possible for the method to simply block waiting for its own data. This is because the method must return before anything else can execute.

To deal with this, the member functions "`suspendResponse()`" and "`resumeResponse()`" are provided. If the "`suspendResponse()`" member function is called, it will raise an exception which will be caught by the service agent framework. The name of this exception is `DelayedResponse` and lets the service agent framework know that a response will be sent at a later time.

When the member function "`suspendResponse()`" is called, a callback function should be supplied as argument which finalises the request and returns the appropriate result. The callback passed to "`suspendResponse()`" will only be called when the "`resumeResponse()`" method is called at some later point in time. In particular, you would call "`resumeResponse()`" once you have collected together all the data which forms the result for the original request.

```
class DatabaseProxy(netsvc.Service):
  def __init__(self,name="database-proxy")
    netsvc.Service.__init__(self,name):
    self.exportMethod(self.tablesRequest,"tables")
    self._request = {}
    self._result = {}
  def tablesRequest(self):
    service = self.serviceEndPoint("database")
    id = service.execute("show tables")
    self.processResponse(self.queryResponse,id)
    self.processFailure(self,queryFailure,id)
    self._request[id] = self.conversationId()
    self.suspendResponse(self.tablesResult)
  def tablesResult(self):
    request = self.conversationId()
    result = self._result[request]
    del self._result[request]
    return result
  def queryResponse(self,id,result):
    if self._request.has_key(id):
      request = self._request[id]
      self._result[request] = result
      del self._request[id]
      self.resumeResponse(request)
  def queryFailure(self,id,error,description,origin,details):
    if self._request.has_key(id):
      request = self._request[id]
      del self._request[id]
      self.cancelResponse(request,error,description,origin,failure)
```

As can be seen, it will be necessary to save away state information about a suspended request so it can be later resumed. In this example the conversation id of the original request is associated with the conversation id of the downstream request. When the result of the downstream request is received, it can be saved away and the original request resumed with the cached result being returned. In the event that the downstream request fails, the "`cancelResponse()`" method is used to abort the original request.

As with the "`abortResponse()`" member function, if "`suspendResponse()`" is being called from within a method, it will be necessary for any code to be explicit about what exceptions it catches, or to at least catch the `DelayedResponse` exception and pass it on as is.

Note that "`suspendResponse()`" and "`resumeResponse()`" were only added in OSE 7.0pl5 and are a layer on top of the "`delayResponse()`" method which only performed the single operation of raising the exception of type `DelayedResponse`. The newer functions should make the task of implementing a delayed responese easier, so if you are using "`delayResponse()`" you should change your code to use the newer functions.

## Identity of the Sender

Normally it is not necessary to know the identity of the sender of a request. If a means of identifying who has initiated the request is required however, the details of the current request can be queried to obtain the address of the sender. This can be useful where a separate session object in the form of a new service is created to manage interaction with a particular client. To obtain the request object for the current request the "`currentRequest()`" member function is used.

By calling the "`sender()`" member function of a request object, the service address of the service agent initiating the request can be obtained. Having created a separate session for that client, all requests for that session can be authenticated as being from the same service agent. Such a scheme may even have as a prelude a log in mechanism to ensure that a service agent making the request has sufficient privileges to initiate a separate session.

Whether or not a login and password is required, the idea is that the method used to initiate the session returns the name of the service created to manage the session. Such a session object should monitor the existence of the service agent who initiated the session such that the session can be destroyed automatically when the owner is withdrawn.

```
class Session(netsvc.Service):
  def __init__(self,name,client):
    netsvc.Service.__init__(self,name)
    self._client = client
    self.subscribeServiceAddress(self.announce,client)
    self.exportMethod(self.close)
  def announce(self,binding,status):
    if status = netsvc.SERVICE_WITHDRAWN:
      self.destroyReferences()
  def close(self):
    client = self.currentRequest().sender()
    if client != self._client:
      self.abortResponse(1,"Not Owner of Session")
    self.destroyReferences()

class Service(netsvc.Service):
  def __init__(self,name="service"):
    netsvc.Service.__init__(self,name)
    self._count = 0
    self.exportMethod(self.login)
  def login(self,login,passwd):
    # authorise login/passwd
```

```
        client = self.currentRequest().sender()
        self._count = self._count + 1
        name "%s/%d" % (self.serviceName(),self._count)
        session = Session(name,client)
        return name
```

Such a mechanism as described can't be used if such a request to create a session may originate over an RPC over HTTP connection. This is because the service agent which acts as proxy for the request is transient and will be destroyed once the request has completed. Further, the service agent which acts as proxy isn't visible outside of its own process.

The alternative to binding the session to a particular service agent is to create a pseudo unique name for the service managing the session. To ensure that the session object is destroyed, a timer could be used to trigger the destruction of the service after a certain period of inactivity. Each request made against the service would reset the timer giving it a new lease on life. The timeout may be something which is fixed or which could be defined as one of the arguments supplied in the request to create the session.

## Invalid Request Method

When a service request arrives with a method name which the service doesn't provide, the member function "`invalidMethod()`" is called. By default this method will generate a failure response with origin of "`netsvc`" and error code of "`SERVER_METHOD_UNAVAILABLE`". This member function might be overridden if the ability to dump out information about requests against invalid methods was wanted. Any derived implementation of this member function should still call the base class version to generate the appropriate failure response indicating a method was unavailable.

```
class Service(netsvc.Service):
  # ...
  def invalidMethod(self,methodName,params):
    print methodName,params
    netsvc.service.invalidMethod(methodName,params)
```

## Local Service Requests

Use of the interface described so far for initiating a service request is the preferred interface. This is because it will not matter if the service to which the request is being sent is in the same process or another process. It also will not matter if the service is written in the same language or a different language. However, when the service is in the same process and is also written in Python a short cut is available. This will avoid the complexity of using delayed responses, but does mandate that the service being called always be in the same process.

Access to this short cut is through the Python class `LocalService`. An instance of the class is created with the name of the service against which the call is to be made. A call is then made against the object as if it were the actual service. This is the same as when a service endpoint object is used except

that the result is returned immediately. Note that since the response is immediate, you can't call a method which itself would try and use a delayed response.

```
class DatabaseProxy(netsvc.Service):
  def __init__(self,name="database-proxy")
    netsvc.Service.__init__(self,name):
    self.exportMethod(self.tables)
    self._active = {}
  def tables(self):
    service = netsvc.LocalService("database")
    return service.execute("show tables")
```

As with a service endpoint object, if there is more than one service agent with the same name, the first one found will be used. The only restriction is that candidate service agents will only come from the set of service agents in the same process which are also implemented in Python. If a request is made against an instance of `LocalService` and no service agent could be found, an exception of type `ServiceUnavailable` will be raised. To avoid this, you can also perform a truth test against the object.

Although the request is channelled through directly to the service instance, it is still not possible to call methods of the service which haven't been exported. When this occurs, an exception of type `ServiceFailure` is raised where the origin is set to "`netsvc`" and the error code is set to "`SERVER_METHOD_UNAVAILABLE`". Any other errors raised by the method being called are similarly indicated using the `ServiceFailure` exception. Note that each of the attributes of the failure, ie., the error code, description, origin and details, are available using member variables and not member functions as is the case with a failure response object.

If the member function making the request is servicing a request from another service, it may be appropriate to translate the exceptions into different types of failure responses. As is, the exceptions would translate into a failure response with the same details. This may be confusing for example if it were an exception indicating that a method was unavailable. The remote service making the request would errornously think that it had called an invalid method when it was actually the implementation of the method which it had called which had done the wrong thing.

Note that the LocalService class is being deprecated and will most likely not be available in a future version of OSE. You are therefore advised not to write any new code using it and change existing code to use the full messaging system features.

# Message Exchange

The features of the service agent framework may be used standalone within a single process or across a set of connected processes. That is, use of the service agent framework is not dependent on a process being able to connect to a central message exchange process. When combined with the HTTP servlet framework and RPC over HTTP interface, a single process may be more than adequate for many applications, especially simple web based services.

If such a service starts to out grow the bounds of a single process however, the application can easily be split up across multiple processes or machines. This will enable services to be distributed based on load or proximity to required resources. Being able to split up the application in this way is also advantageous in that it becomes easier to introduce into the application distinct components which are written in C++ as opposed to Python.

Unlike most message oriented middleware packages, there is no dedicated message exchange process. Instead, the components relating to client and server aspects of the mechanism for implementing a distributed service agent framework are directly accessible. This means that it is possible to take an existing application and embed within it a message exchange server endpoint. Growing your application then becomes a simple matter of creating new processes which incorporate a message exchange client endpoint and have it connect to your original application.

The major classes in the OSE C++ class library used to provide this functionality are the `OTC_Exchange`, `OTC_InetClient` and `OTC_InetListener` classes. Note that the Python interface only provides the ability to create connections between processes which make use of the INET socket protocol. When the C++ interface is used directly, on a UNIX platform there is also the option of using the UNIX socket protocol.

## Exchange Initialisation

To create a message exchange endpoint in a process, the `Exchange` class is used. When creating an instance of the `Exchange` class it is necessary to specify whether it is performing the role of a message exchange server or that of a client. A message exchange server is a process which takes on the role of being a hub for message distribution. That is, a message exchange server is a process which accepts connections from one or more message exchange clients and distributes messages between the client processes as appropriate.

Two different approaches can be taken in regard to the message exchange server. The first is that the message exchange server component can be embedded within an existing application and new clients attach to that existing application. Alternatively, a separate process can be created which embeds just the message exchange server component and the existing application, now modelled as a client, along with any new clients connect to this new process.

In both server configurations, initialisation of the message exchange server endpoint is the same. Subsequent to initialisation, the endpoint is then directed to listen on a specific port for any client connections.

```
port = 11111
exchange = netsvc.Exchange(netsvc.EXCHANGE_SERVER)
exchange.listen(port)
```

In the case of a message exchange client, instead of listening for connections, the endpoint is directed to connect to a message exchange server.

```
host = "localhost"
port = 11111
retry = 5
exchange = netsvc.Exchange(netsvc.EXCHANGE_CLIENT)
exchange.connect(host,port,retry)
```

Because it is possible that the message exchange server is not available, a retry delay can be specified. When supplied this will result in successive attempts to connect to the server until a connection is established. The retry delay when supplied needs to be specified in seconds.

Note that if a connection to the server is lost, the client will also attempt to reconnect automatically after the retry delay time has expired. This has the affect that a client will always try to stay connected to its server without you needing to take any specific action. Your process will not be prematurely shutdown if a connection cannot be established or if a connection is lost.

## Service Availability

Unless the service audience of a service agent has been set so as to restrict its visibility, a service will automatically become visible within connected processes through the service registry of the remote process. That is, if a particular service is located within the same process as the message exchange serv-

er endpoint and a new client connects, a subscriber to that service in the client will be notified that the service is available. Similarly, any service within a client will become visible from the server as well as other connected clients.

Although the service is located in a separate process, the same service registry interface is used to subscribe to the presence of the service. Subscription to reports produced by the service and the issuing of requests against that service are also mediated through the same interface as previously described. The only exception to this is that the `LocalService` proxy class cannot be used to communicate with any service in a remote process, it being restricted to services implemented using Python which appear in the same process.

Except for the `LocalService` proxy class, that there is no distinction in the interface to communicate between services whether they be in the same or a remote process, means that it is a simple matter to split an application across multiple processes. If a distinct message exchange server process is used, all that is required is that each process embed a message exchange client and connect to the message exchange server.

As an example, a process supporting a service which publishes periodic reports would be written as follows.

```
class Publisher(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__("publisher")
    self.publishReport("system.ctime",netsvc.DateTime(),-1)
    self.startTimer(self.timeout,10,"heartbeat")
  def timeout(self,tag):
    self.publishReport("system.time",netsvc.DateTime())
    self.startTimer(self.timeout,10,"heartbeat")

dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)
exchange = netsvc.Exchange(netsvc.EXCHANGE_CLIENT)
exchange.connect("localhost",11111,5)
dispatcher.run()
```

The process containing the corresponding subscriber to this service would then be written as follows.

```
class Subscriber(netsvc.Service):
  def __init__(self):
    netsvc.Service.__init__(self)
    self.monitorReports(self.report,"publisher","system.*")
  def report(self,service,subject,content):
    name = service.serviceName()
    identity = service.agentIdentity()
    publisher = "(%s/%s)" % (`name`,identity)
    if subject == "system.ctime":
      now = str(netsvc.DateTime())
      print "%s became available at %s" % (publisher,now)
```

```
        print "%s originally started at %s" % (publisher,str(content))
      elif subject == "system.time":
        print "%s was still alive at %s" % (publisher,str(content))

dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)
exchange = netsvc.Exchange(netsvc.EXCHANGE_CLIENT)
exchange.connect("localhost",11111,5)
dispatcher.run()
```

The only difference is that a message exchange client has been added to each, the actual services are identical to what they were when used in the same process.

In regard to announcements of service availability and their subsequent withdrawal, when everything is in the same process, such an announcements means that the service had been created or destroyed. In the context of a distributed system, such an announcement means that a service is now visible or is no longer visible. Such an announcement doesn't mean that the service was necessarily destroyed as it could be the case that the message exchange server process was shutdown. Thus the service could still exist, it just may not be reachable.

Because services may become unavailable, or connections lost and also because connections between processes will automatically restart when possible, it is important that client services take notice of announcements regarding the availability of a service it is using. A client service should not assume that a service it is using will always be available and should be programmed to accommodate this fact.

## Connection Announcements

Monitoring the existence of services gives precise information about when such services become available. This however may be too much fine detail. If a client process needs to merely know when a connection had been established to the message exchange server, it is possible to create a derived version of the `Exchange` class and override the "`handleConnection()`" member function.

This member function will be called when a client has successfully connected to a server, when that connection is subsequently lost, or when an initial connection attempt fails. On the server side, the member function is called when a connection is accepted and when it is lost.

```
class Exchange(netsvc.Exchange):
  def __init__(self,type):
    netsvc.Exchange.__init__(self,type)
  def handleConnection(self,announcement):
    state = "INACTIVE"
    if announcement.state() == netsvc.CONNECTION_ACTIVE:
      state = "ACTIVE"
    process = announcement.remoteProcess()
    address = announcement.remoteAddress()
    message = "%s %s (%s)" % (state,process,address)
    logger.notify(netsvc.LOG_NOTICE,message)
```

Overriding this method can be useful purely for logging purposes, but might also be used in a client process to trigger an announcement to activate the function of the process upon a connection becoming active. Consequently, the operation of a client process could be suspended or the process shutdown when no active connection could be established or the connection lost.

This latter mode of operation would be necessary when a retry delay is not specified when connecting a message exchange client to a server. In this situation the retry delay defaults to the value of "`-1`", indicating that one and only one connection attempt should be made. If this is used, a client should monitor to see if the connection fails and shutdown the process if it does. Similarly, if it does manage to connect to the server, when that connection is subsequently lost the process should again be shutdown.

Note that creation of a one off connection will currently consume resources that cannot be reclaimed. This is a limitation of the Python interface and is not present when using the OSE C++ class library directly which has a way of reclaiming the resources. As the intent is that the message exchange framework is for permanent connections, this is not seen as too problematic at this time and will only be addressed at some time in the future.

## Authorisation of Clients

As the message exchange framework provides direct access into an application, it may be desirable to restrict which hosts can connect in to an application. If this type of control is required, it can be implemented by creating a new derived version of the `Exchange` class and overriding the member function "`authorise()`". For each client connection that a server gets, this member function will be called with the IP address of the host the client is located on. A server may then reject or accept the connection.

```
class Exchange(netsvc.Exchange):
  def __init__(self,type,hosts=[]):
    netsvc.Exchange.__init__(self,type)
    self._allow = hosts
  def authorise(self,host):
    return host in self._allow
```

To accept a connection the member function should return a true value and false otherwise. When a connection is rejected, the client will see it as a failed connection attempt.

## Distributed Exchange Server

When an application is distributed across multiple machines, it may not be desirable that processes on one machine must connect to the message exchange server located on another machine. The problem here is that if the machine hosting the message exchange server is shutdown, none of the processes located on remote machines will be able to communicate with each other. In essence there is a single point of failure.

When an application is distributed across multiple machines, it is often the case that even if one machine were to be shutdown, the processes on a different machine might be able to quite happily keep operating so long as they could still communicate. To support this, a means of setting up a distributed version of the message exchange server is provided.

In this arrangement, each machine has its own message exchange server, with each message exchange server connected to all others. If a machine is now shutdown or connections to one machine lost, other machines will still be able to communicate with processes on any machines which are still accessible. That is, loss of the message exchange server on one machine will only directly impact that machine.

To setup a distributed exchange server, the message exchange server endpoint is created as before. The difference is that as well as listening on a port for new connections, client like connections are created to the other message exchange servers. The aim here is to effectively create a star connected network between the message exchange servers. That is, each message exchange server has a connection to all other message exchange servers.

```
port = 11111
exchange = Exchange(netsvc.EXCHANGE_SERVER)
exchange.listen(port)

delay = 5
for host in hosts:
  exchange.connect(host,port,delay)
```

Note that since connections are bidirectional, it is not necessary for each message exchange server to mutually connect to each other. That is, if you have two message exchange servers, it is only necessary for one to connect to the other. In other words, the list of remote hosts in one would be empty, where as the list of the remote hosts in the other would be the reciprocal host. If two message exchange servers do connect to each other, this will be detected and one connection will be ignored, however it should be avoided.

## Multiple Exchange Groups

When creating a service agent, the default service audience is "*", indicating that knowledge of the service should be distributed as widely as possible. One alternative is to set the service audience to the empty string, which will always result in the service only being visible within its own process. What occurs for other values of the service audience property depends on the exchange group assigned to a message exchange endpoint.

By default, the exchange group of a message exchange endpoint is empty, but may be set by an optional argument when initialising the class. A message exchange endpoint is only able to be connected to a complimentary message exchange endpoint which is a member of the same group. That is, a message exchange client endpoint can only connect to a message exchange server endpoint with the same exchange group.

With respect to service visibility, a message exchange endpoint will only pass information about services if the service audience is "*", or if the service audience is the same as the exchange group. The only exception to this is when the exchange group is empty. In that case, an empty service audience will still restrict visibility of a service to its own process.

By using multiple exchange groups within an application, it becomes possible to segment an application into parts and restrict visibility of services to those parts of the applications which need to see them. As an example, a service may act as a front end for multiple back end services which do the real work and for which it is not necessary that they be visible.

In this example, the process containing the front end service, as well as creating a message exchange client endpoint for the default exchange group, would create its own message exchange server endpoint. The default name for this exchange group would be overridden and a different port used for connections. Back end processes would then connect to this new port, with all services in the back end processes having a service audience matching that of the new exchange group.

```
class FrontEnd(netsvc.Service):
  def __init__(self,name="database")
    netsvc.Service.__init__(self,name)
    self.subscribeServiceGroup(self.announce,"backend")
  def announce(self,binding,group,status):
    if binding.serviceAudience() == "database":
      # this is one of ours

default = netsvc.Exchange(netsvc.EXCHANGE_CLIENT)
default.connect("localhost",11111,5)
backend = netsvc.Exchange(netsvc.EXCHANGE_SERVER,"database")
backend.listen(11112)
```

The front end service would use subscription to a service group to know about the existence of any back end services. Each of the back end services would in turn add themselves to the same group so the front end is aware of their existence. The front end service can check the service audience for a service to know for sure that it is one of its back end services and not an imposter visible through the default exchange group.

```
class BackEnd(netsvc.Service):
  def __init__(self,name="",audience="database")
    netsvc.Service.__init__(self,name,audience)
    self.joinGroup("backend")

backend = netsvc.Exchange(netsvc.EXCHANGE_CLIENT,"database")
backend.connect("localhost",11112,5)
```

Having done this, any services within the back end process will only be visible from other back end processes and the front end process. The services in the back end process will not be visible within any process reachable from the front end process over the original message exchange client endpoint at-

tached to the default exchange group. Back end services will still be able to see any services on the default exchange group which had a service audience of "*".

Note that different exchange groups should not overlap. That is, they should only ever share at most one process with any other exchange group. In effect, exchange groups when used should form a hierarchy. The only time that loops are allowed within the way processes are connected is when creating a distributed exchange server for a specific exchange group.

## Scalability of the Framework

Because there is no dedicated message exchange server process serving as the sole repository of service information, the service registry in each process will contain a record of all services it can see. As the size of an application grows to have very large number of services this may result in the size of what otherwise should be a small process to grow unnecessarily.

Currently there are couple of approaches that can be taken to reduce this problem, however, it is recommended that if you know that you will have very large numbers of services and specifically publishers and subscribers, that you might be better off purchasing one of the commercial products which are specifically designed and targeted at such large scale systems. Such products might not support the concept of distinct services and instead implement a flat name space for subscriptions, but they are more likely to scale better.

In other words, the design of the service agent framework and the message exchange framework lends itself to small to medium size systems. Don’t expect to be able to run the whole of the New York stock market data feeds through this system as it will more than likely not suit your requirements.

Having made this disclaimer, the first things you can do to reduce growth in the size of the service registry in each process, is to not export a service beyond the scope of a process unless you really need to. That is, if the service only needs to be visible within its own process, sets its service audience to be the empty string.

Such a service will not be visible outside of the process and that service will not be able to subscribe to services outside of the process, but in most cases the service will still be able to make a request against a remote service. Restricting the visibility of a service to its own process will also cut down on traffic between processes relating to the existence and withdrawal of services.

The next thing which can be done is to look more closely at the relationship that exists between services. If there are a group of related services which only need to talk to each, locate them together. This can be done by putting them in the same process and restricting visibility to that process, or by separating them from the remainder of the application by creating a distinct exchange group. In both cases, have only the services which need to be public actually visible globally.

# Message Encoding

As the service agent framework is designed as a distributed system covering multiple programming languages, it is necessary that any data being passed around within a report, request or response be serialised into a form suitable for transmission as part of a message. At present the encoded form of the data uses a subset of XML. That is, it would qualify as being XML, however to make the implementation easier, the code for decoding such messages will not accept arbitrary XML.

At present the exact form of the XML being used is not revealed as this is being reviewed and will most likely change. Further, the protocol used between message exchange endpoints is unique to this software. It too is being reviewed and will most likely be changed to use some more commonly accept form of handling message boundaries. Any new mechanism will likely also be designed to be able to proxy through HTTP servers, thus avoiding issues with closed firewalls.

That the precise details are not being revealed actually makes no difference as it has no bearing in relation to using the software. This is because everything is hidden under a high level API which hides such details, thus allowing for change in the formats used without requiring changes to applications using the software. The only instance where changes might have a visible affect is in respect to the NET-RPC protocol for RPC over HTTP. This would only be an issue if you tried to write your own client for this protocol.

The one aspect of how data is encoded which will not change is in relation to the means of identifying different types. Here the XML Schema Datatypes 2001 specification is used as a guide, with Python types being assigned corresponding types with respect to this specification. Through introduction of customised encoders and decoders, support for user defined scalar data types may however also be added.

## Supported Data Types

Communication between services is mediated through a layer of code which is written in C++. The only exception to this is when the `LocalService` class is used as a proxy to send a request to a service in the same process which is also implemented in Python. This means that except for when the `LocalService` class is being used, any data which is being transferred between services must go through a process of being encoded into a serialised form at the point of sending and then deserialised at the point of reception.

Data which is being sent between services is not limited to that of just a string. The data to be sent can consist of any of the basic Python scalar types, a list, a tuple or a dictionary. In addition to this, the Python `None` value may be used, as well as a number of extended types. The only limitation in respect of the Python compound types is that when using a dictionary, the keys must be of type string. Further, when a tuple appears within any data, the recipient will see it as a list and not a tuple. It is not possible to send data which is cyclically self referential.

```
self.publishReport("string","value")
self.publishReport("list",[1,1L,1.1,None])
self.publishReport("dictionary",{"key":"value"})
```

The extended types which are supported are `Boolean`, `Binary`, `Date`, `DateTime`, `Time` and `Duration`. For the `Boolean` type, there are also predefined values for `True` and `False`. The `Boolean` type should behave correctly with respect to all truth type tests. If the default arguments for the constructor of `Date` and `DateTime` types are used, they will be initialised to the current local date and current local date and time respectively.

```
self.publishReport("true",netsvc.True)
self.publishReport("false,netsvc.False)
self.publishReport("boolean",netsvc.Boolean(1))

self.publishReport("binary",netsvc.Binary("value"))

# current local date
self.publishReport("date",netsvc.Date())

# current local date/time
self.publishReport("dateTime",netsvc.DateTime())
```

When using the various date and time types, they should be initialised with string values corresponding to what type they represent. The format and range of these values should be the subset of values possible under the ISO 8601 date/time standard as described by the XML Schema Datatypes 2001 specification, examples of which are illustrated below.

| Type | Format | Sample |
|---|---|---|
| `Date` | YYYY-MM-DD | 2001-12-25 |

| Type | Format | Sample |
|---|---|---|
| `DateTime` | YYYY-MM-DDThh:mm:ss | 2001-12-25T23:59:59 |
| `Time` | hh:mm:ss | 23:59:59 |
| `Duration` | PnDTnHnMnS | P1DT23H59M59S |

For the date and time types, the current Python implementation does not do any checking to determine if the supplied values are valid, but will pass them as is. Note that the XML Schema Datatypes specification does allow for a timezone in a date and time, but it is recommended that all date and time values be sent as UTC. In the C++ library, only classes corresponding to `Date` and `DateTime` exist. These are `OTC_Date` and `OTC_Time`. The `OTC_Time` class is not able to handle timezones.

The only difference between the `Binary` type and using a string is that the value supplied via the `Binary` type, will be encoded internally using "base64" encoding when being passed around. This has relevance because in XML most control characters are not permitted in string values. An XML implementation can also collapse a "`\r\n`" combination to just "`\n`". If such characters may appear in a string, you should use the `Binary` type to ensure that they are preserved as is. Note that you do not however have to encode the string using base64 encoding first as the internal implementation will do this for you automatically.

## Mapping of Scalar Types

When data is being serialised, the names attributed to scalar types derive from the XML Schema Datatypes 2001 specification. The only exception to this is the `None` type, which notionally is passed around internally with an empty type value. The mapping from Python types to those described in the XML Schema Datatypes specification is as follows.

| Python Type | Encodes To XML Type |
|---|---|
| `string` | `xsd:string` |
| `int` | `xsd:int` |
| `long` | `xsd:long` |
| `float` | `xsd:double` |
| `netsvc.Boolean` | `xsd:boolean` |
| `netsvc.Binary` | `xsd:base64Binary` |
| `netsvc.Date` | `xsd:date` |
| `netsvc.DateTime` | `xsd:dateTime` |

| Python Type | Encodes To XML Type |
|---|---|
| `netsvc.Time` | `xsd:time` |
| `netsvc.Duration` | `xsd:duration` |

If a service is implemented using the OSE C++ class library directly, different size versions of the integer and floating point types are available and can be generated in the serialised form of any data. A consequence of this is that when converting any data from its serialised form into instances of Python types, a broader range of possible values types need to be accommodated.

| XML Type | Decodes to Python Type |
|---|---|
| `xsd:string` | `string` |
| `xsd:byte`<br>`xsd:short`<br>`xsd:int`<br>`xsd:unsignedByte`<br>`xsd:unsignedShort`<br>`xsd:unsignedInt` | `int` |
| `xsd:long`<br>`xsd:unsignedLong`<br>`xsd:integer,` | `int` or `long` as appropriate |
| `xsd:float`<br>`xsd:double`<br>`xsd:real` | `float` |
| `xsd:boolean` | `netsvc.Boolean` |
| `xsd:base64Binary` | `netsvc.Binary` |
| `xsd:date` | `netsvc.Date` |
| `xsd:dateTime` | `netsvc.DateTime` |
| `xsd:time` | `netsvc.Time` |
| `xsd:duration` | `netsvc.Duration` |

Note that at the present time, not all of the XML data types in respect of non positive and non negative integers are accommodated. These will most likely be added at some time in the future, however in the short term they don’t add anything extra in relation to the Python interface. Support for the type

"xsd:hexBinary" will also be added at some point in the future as well. If you wish to send a Unicode string, you should convert it into a string using UTF-8 encoding.

## User Defined Types

The intent with the XML Schema Datatypes specification is that additional scalar data types can be introduced by assigning a new name scoped within a distinct namespace. In respect of the types defined by this specification, the namespace "xsd" is used. Note that within this implementation, the namespace is not linked to a URI containing any form of definition for that type. If sending a data value of your own type, it is up to your code to ensure that both ends know what the type means.

The simplest way of adding your own types is by using the Opaque class. When initialised this takes two values, a string identifying the type of value and a string representing the value in its encoded form. It is not necessary to escape any characters in the encoded value which may be special to XML as such values will be automatically escaped as necessary.

```
data = complex(1,1)
type = "python:complex"
self.publishReport("complex",netsvc.Opaque(type,data))
```

In reality, it isn't actually necessary to encode a Python complex value in the way shown as a special mapping is by default installed for this type. For this Python type the namespace "python" is used. If defining your own type it is recommended you use some other namespace value which is in some way specifically associated with your application or some third party standard relating to additional XML types.

As a special mapping is provided for the Python complex type, it will be decoded into an instance of the Python complex type on reception. If however a mapping is not available for a specified type, the value will be converted back into an instance of the Opaque type. The type associated with the value can then be queried using the "type" attribute and the actual encoded data using the "data" attribute.

```
def dump(self,object):
  if isinstance(object,netsvc.Opaque):
    print object.type,object.data
```

The Opaque class provides a means of sending a value without a defined mapping, or of you being able to receive values for which no mapping is defined. If necessary the interface of the Opaque class can be used to dynamically handle such unknown values and perhaps still make some sense of them.

## Adding New Mappings

Mappings for new types can be added at two levels. These are at global scope or such that they only apply within the scope of a single service. If a type mapping is added at global scope, you should realise that such a mapping will be applied to any service. Adding new mappings with global scope should therefore be carefully considered as it may inadvertently affect the operation of another service.

To add a new mapping at global scope the functions "`encoder()`" and "`decoder()`" should be used to register functions to do the appropriate conversions. When registering the encoder, the first argument should be either the type object or class object as appropriate. When registering the decoder, the first argument should be the qualified name you have given the type.

The encoder function which you register should accept a single argument, that being an instance of your type. The function should return a tuple containing the qualified name you have given the type and the value encoded as a string. The decoder function should accept two arguments, they being the qualified name you have given the type and the value encoded as a string. The function should return the corresponding instance of the type as described by the encoded value. If the encoded value is invalid, the function should raise an appropriate exception.

```
def _encode_Complex(object):
  return ("python:complex",repr(object))

def _decode_Complex(name,string):
  object = eval(string,{},{})
  if type(object) != types.ComplexType:
    raise TypeError("invalid encoding for complex type")
  return object

netsvc.encoder(types.ComplexType,_encode_Complex)
netsvc.decoder("python:complex",_decode_Complex)
```

To define a mapping which applies only within the context of a single service, you need to override the member functions "`encodeObject()`" and "`decodeValue()`" as appropriate. Note that the default implementations of these methods will apply any global mappings which are present. If your version of these functions, don't identify the type you are interested in, your function should call the base class version of the function. The arguments to these functions are similar to the global encoders and decoders.

```
class Database(netsvc.Service):
  def __init__(self,name,**kw):
    netsvc.Service.__init__(self,name)
    # ...
  def encodeObject(self,object):
    if hasattr(MySQLdb,"DateTime"):
      if type(object) == MySQLdb.DateTimeType:
        return ("xsd:string",object.strftime())
      elif type(object) == MySQLdb.DateTimeDeltaType:
        return ("xsd:string",str(object))
    return netsvc.Service.encodeObject(self,object)
```

Providing a mapping which is specific to a service is most often used when the service interacts with a Python module which defines its own types for such values as date and time. In this circumstance, the mapping function can automatically translate an instance of the type into a type appropriate for the encoded data. This avoids your own code having to manually translate values into corresponding val-

ues of the correct type before hand. A service may also override the default decoders for extended types such as the date and time types if desired.

## Handling Structured Types

The encoding mechanism for data does not provide a way of adding support for your own structured types, whereby the type of that object can also be transmitted. All objects need to be able to be converted into instances of scalar types, dictionaries, tuples or lists. To avoid having to do this conversion manually, it is however possible to define an encoder for a structured type which will do this for you.

At the global level, such a function is again registered using the "`encoder()`" function. The difference between this function and that for scalar types however, is that instead of returning a string giving the name of the scalar type, the value `None` should be returned in its place. The second value in the tuple should then be the instance of the structured type translated into either a scalar type, dictionary, tuple or list.

```
def _encode_UserList(object):
  return (None,list(object))

netsvc.encoder(UserList.UserList,_encode_UserList)
```

Having returned the translated value, it will be represented to the encoder. Thus it is only necessary to translate the top level of the data structure as enclosed values will in turn be translated automatically if required and if an encoder is registered. This mechanism may also be used in an encoder specific to a service.

# Servlet Framework

The HTTP servlet framework can be used to provide a window into your application. A number of predefined servlets are provided or you can create your own. You can also create your own server objects to map the servlets to appropriate parts of the URL namespace. Alternatively, a number of predefined server objects can be used for common tasks such as serving up files from the filesystem, or provision of RPC over HTTP services. Basic user authentication is implemented and clients can also be blocked based on their address.

The major classes in the OSE C++ class library involved in providing this functionality are the `OTC_HttpDaemon`, `OTC_HttpServer` and `OTC_HttpServlet` classes, plus the various derived servlet and server classes. The implementation of the HTTP servlet framework is based on the event system and multiple HTTP requests can be handled concurrently.

Although the framework is quite powerful, you should still keep in mind that its main purpose is for interacting with an application. If you are after a general purpose web server, you would probably be better off using a product like Apache. If you need the appearance that the web site and application are one, use the "`mod_proxy`" plugin for Apache to redirect only a portion of the URL namespace. This actually has the added benefit that Apache can be used to setup a SSL connection with the client over any insecure network, with communication between Apache and the application on the secure network being normal HTTP.

## Framework Overview

When a HTTP client makes a connection to the server process, a session manager is created to parse any requests made by that client. For each request, an attempt is made to find a server object which manages the part of the URL namespace that the request falls under. This server object is then asked

to provide a servlet to handle the actual request. If no server object is found corresponding to that portion of the URL namespace, or the server object is not able to provide a servlet to handle the request, a HTTP error response is returned to the client indicating that the resource corresponding to the supplied URL could not be found.

Where an appropriate servlet to handle the request is found, the session manager will initially pass off to the servlet the details of the request. This will include the type of request, the URL and the contents of any HTTP headers. The details initially provided to the servlet do not include any content associated with the request. Any content associated with a request will subsequently be passed to the servlet as it arrives. This will only occur though if the servlet wasn't able to process the request based on the initial information and does actually require the content.

In the majority of cases a request will not have any associated content and a servlet will be able to process the request straight away. Even if there is no content however, the servlet isn't obligated to send a response immediately. This may be the situation if the servlet needs to wait until information from another source arrives before it can form the response. In this scenario, the servlet might send a request using the messaging framework to a remote service to obtain the information. When the response from the remote service arrives, the servlet can then generate the response.

When the action of the servlet does depend on the content supplied with the request, the servlet would accumulate the content as it arrives until the amount of content matches that given in the content length header, or until some appropriate boundary is encountered. Now having all the content associated with the request, the servlet can process the request and send a response. Alternatively it could again delay the response if it needs to first send the content received to some remote service and wait for some response.

When a servlet sends a response to the HTTP client, as long as the servlet generates a content length in the HTTP headers, any request by a HTTP client to keep alive the session will be honoured. This allows the HTTP client to submit additional requests using the same connection if desired. In general the servlet framework adheres to the HTTP 1.0 protocol.

## The HTTP Daemon

The Python class which listens for connection requests from HTTP clients is called `HttpDaemon`. When creating the HTTP daemon, you need to tell it which port to listen on and also register with it any HTTP server objects. When registering a HTTP server object, you need to identify which part of the URL namespace it manages. Finally, you need to start the daemon so that once the dispatcher is run it will actually listen and handle the requests.

```
dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)

daemon = netsvc.HttpDaemon(8000)
filesrvr = netsvc.FileServer(os.getcwd())
daemon.attach("/",filesrvr)
```

```
daemon.start()

dispatcher.run()
```

When a HTTP server object is registered, the first argument to the "`attach()`" member function should be the path under which resources made available by the HTTP server object are accessible. Except for the root directory, the path should not include a trailing "`/`". The path should also be in normalised form. That is, it should not include consecutive instances of "`/`" within the path, or include the path components "`..`" or "`.`".

If the path isn't normalised in this respect, these paths will never match against any request as request URLs will always be normalised before attempting a match. The request URL is always normalised to avoid the possibility of malicious requests trying to access file type resources outside the available directory tree.

If desired, a single HTTP server object may be registered multiple times within the one URL namespace. Registrations may also be done hierachically. That is, one registration may nest within the URL namespace of another. In this situation a request will match against the HTTP server object with the most deeply nested path.

```
filesrvr1 = netsvc.FileServer(os.path.join(os.getcwd(),"info"))
filesrvr2 = netsvc.FileServer(os.path.join(os.getcwd(),"logs"))
daemon.attach("/",filesrvr1)
daemon.attach("/logs",filesrvr2)
```

Normally the port which the HTTP daemon is to listen on will be fixed. If you require a dynamically allocated port, you should use "`0`" as the port number. The actual port number which is allocated can then be queried using the "`port()`" member function. Obviously, this port number would then need to be displayed somewhere or otherwise accessible so it is known which port to connect to.

```
daemon = netsvc.HttpDaemon(0)
port = daemon.port()
```

## The File Server

The `FileServer` class is a predefined HTTP server object for serving up files from the file system in response to HTTP GET requests. This server object is suitable for providing access to documentation related to an application, configuration files or application log files. A plugin mechanism for handling special file types is also included

In the case of files resident in the file system, the server is able to handle any size file, with the corresponding servlet only sending data back to the HTTP client as it is able to receive it. That is, transmission of a large file will not blow out the size of the application nor will it cause the application to block if the client is slow at reading the contents of the file.

When an instance of the `FileServer` class is created, it must be supplied with the filesystem directory from which files are to be served. The server object utilises the Python `mimetypes` module for determining file types. The file type associated with an extension can be overridden, or knowledge of additional file types can be added using the "`map()`" member function.

```
filesrvr = netsvc.FileServer("/home/httpd")
filesrvr.map(".py","text/plain")
```

Note that it is expected that the HTTP client knows the name of the file it is trying to access as there is no builtin support included for directory browsing. It is however possible to define the names of one or more index files to try when a request identifies a directory as opposed to a file. When more than one index file is specified, those which were declared later, take precedence.

```
filesrvr.index("index.htm")
filesrvr.index("index.html")
```

Editor backup files, temporary files generated by an application, or any other files which should not in any way be accessible from a HTTP client, can be hidden from view so long as they have a distinct extension.

```
filesrvr.hide(".bak")
filesrvr.hide(".html~")
```

When it comes to the actual task of serving up a single file from the file system, the `FileServlet` class is used. This is a wrapper around the corresponding servlet class from the OSE C++ class library used to handle the request for a single file. The servlet may be used directly from a custom HTTP server object.

## Client Authorisation

If the HTTP servlet framework is being used to provide an administrative interface into an application, it may be desirable to block access from all but a few selected client hosts. This can be useful where the application is otherwise intentionally accessible over the Internet, or may inadvertantly become accessible from a broader range of hosts than intended. This may result from misconfigured firewalls, or the addition of additional subnets to a corporate network.

If you wish to control who can access the application through the port monitored by the HTTP daemon, it is necessary to create a derived version of the `HttpDaemon` class and override the "`authorise()`" member function. For each client connection, this member function will be called with the IP address of the client host. Your code can thereby block requests from any undesirable hosts.

```
class HttpDaemon(netsvc.HttpDaemon):
  def __init__(self,port,hosts=[]):
    netsvc.HttpDaemon.__init__(self,port)
    self._allow = hosts
  def authorise(self,host):
    return host in self._allow
```

If access to a particular client is disallowed, the connection will be dropped immediately. The client will not receive any form of specific HTTP error response indicating why the connection has been closed. Note that this mechanism blocks a client from accessing any part of the URL namespace for that HTTP daemon. If you wish to only block client access to specific resources, you would need to customise each HTTP server object or servlet, or use multiple HTTP daemon objects on separate ports.

## User Authorisation

A further level of authorisation beyond that of blocking specific client hosts is to individually authenticate each user. The mechanism for user authentication is performed against the HTTP server objects. That is, the URL namespace managed by each HTTP server component can be individually protected using different user databases.

To add user authentication to a particular HTTP server object, you should derive from the class and override the "`authorise()`" member function. If building your own HTTP server object, you could embed the member function directly in your class.

```
class FileServer(netsvc.FileServer):
  def __init__(self,directory,users={}):
    netsvc.FileServer.__init__(self,directory)
    self._allow = users
  def authorise(self,login,password):
    return self._allow.has_key(login) and \
        self._allow[login] == password
```

If you need to control user access at the level of individual URLs within the URL namespace managed by a particular HTTP server object, that functionality would need to be embedded into any servlets created by that HTTP server object, or managed at the point that the servlets are created by the HTTP server object. Note that only the HTTP basic authentication mechanism is supported. There is no support for use of secure sockets and SSL.

## HTTP Server Objects

When a HTTP request is received, it is a HTTP server object which will dictate the type of HTTP servlet created to handle the request. If you wish to implement a customised mapping between request URLs and the available HTTP servlets, or introduce a new type of HTTP servlet, you will need to define your own HTTP server object by deriving from the `HttpServer` class and overriding the "`servlet()`" member function.

```
class HttpServer(netsvc.HttpServer):
  def servlet(self,session):
    servletPath = session.servletPath()
    if servletPath == "echo":
      return netsvc.EchoServlet(session)
    elif servletPath == "motd":
      return netsvc.FileServlet(session,"/etc/motd","text/plain")
```

```
        return netsvc.ErrorServlet(404)

daemon = netsvc.HttpDaemon(8000)
server = HttpServer()
daemon.attach("/test",server)
daemon.start()
```

The job of the "`servlet()`" member function is to create an instance of a HTTP servlet capable of handling a request made against a specific URL. When the "`servlet()`" member function is called it is supplied with the HTTP session object. The session object provides access to details of the request, including the server root and servlet path. The server root corresponds to the path under which the HTTP server object was registered. The servlet path is the remainder of the path expressed relative to that server root.

As an example, if the request used the path "`/test/echo`" and the HTTP server object was registered with the path "`/test`", the server root would be "`/test`" and the servlet path would be "`echo`". In the case that a HTTP server object is registered with path "`/`", the server root will still be "`/`". This is the only case where the trailing "`/`" isn't removed.

Under normal circumstances the HTTP server object would determine the type of HTTP servlet to create and the resource being referenced based only on the servlet path. If necessary however, it can query other information related to a request. Such a circumstance might be to look for the presence of cookies used to implement a user session mechanism.

When a HTTP servlet is created, it will need to be passed the handle to the HTTP session object. All the predefined HTTP servlets accept this as the first argument when the servlet is created. If you are defining your own servlets, it is recommended you follow this convention.

If the HTTP server object isn't able to map a request to a particular type of HTTP servlet, the "`servlet()`" member function should return "`None`", or should indicate a specific type of HTTP error response using the `ErrorServlet` class. A HTTP client can be redirected to a different resource using the `RedirectServlet` class.

## The Error Servlet

The error servlet as implemented by the `ErrorServlet` class is provided as a quick way for a custom HTTP server object to return a HTTP error response. In addition to the the HTTP session object, the error servlet needs to be supplied with an approriate HTTP error response code. Text to be included in the body of the response can also be provided if desired. Such text may include any relevant HTML markup, but should not include the opening and closing "`body`" tags.

```
class HttpServer(netsvc.HttpServer):
  def servlet(self,session):
    return netsvc.ErrorServlet(session,501,"Not implemented.")
```

## The Redirect Servlet

The redirect servlet as implemented by the `RedirectServlet` class would be used when it is necessary to redirect a HTTP client to an alternate resource. In addition to the HTTP session object, it should be supplied the URI of the resource to which the HTTP client is to be directed. By default, the HTTP response code will be "`302`", indicating the resource has been temporarily moved. This can be explicitly indicated by using the value "`REDIRECT_TEMPORARY`". If the resource has been permanently moved, the value "`REDIRECT_PERMANENT`" can instead be used.

```
class HttpServer(netsvc.HttpServer):
  def servlet(self,session):
    url = "http://hostname/" + session.servletPath()
    type = netsvc.REDIRECT_PERMANENT
    return netsvc.RedirectServlet(session,url,type)
```

If the URI doesn't start with "`/`", it is assumed to be a valid URI and will be passed as is. If the URI starts with "`/`", it will assumed to be a absolute URL against the current server host and will be automatically adjusted to include the details of the server host in the URL.

## The Echo Servlet

The echo servlet as implemented by the `EchoServlet` class is useful for debugging. When used to service a HTTP request, it will generate a HTML document which provides details about the request.

```
class HttpServer(netsvc.HttpServer):
  def servlet(self,session):
    return netsvc.EchoServlet(session)
```

## The File Servlet

The file servlet as implemented by the `FileServlet` class, is used to deliver up to a HTTP client the contents of a file stored in the operating system's filesystem. This is the same servlet which is used internal to the `FileServer` class. When this servlet is being created it needs to be supplied with the name of the file and the file type. The latter corresponds to the MIME content type included in the HTTP response.

If the path supplied to the `FileServlet` class actually describes a directory, the servlet will generate a response indicating that access is forbidden. If you wish to implement directory browsing you will need to implement a separate HTTP servlet to generate an appropriate response and map the request to it. If you want to redirect the request to an index file, your HTTP server should determine if such an index file exists and if it does, create the file servlet against it instead.

When the `FileServlet` class is used, any size file can be handled without the size of the application growing in size and without the application blocking as a result of a slow HTTP client. This is achieved as a result of the file being sent in blocks, with the servlet waiting if the connection to the HTTP client

becomes congested. Although the servlet may be forced to wait before it can send more data, any other jobs in the event system will still be serviced, including other HTTP requests.

## Logging of Requests

By default no information is logged about requests. If you wish to log what requests are being made against your application using the HTTP servlet framework, you need to set the environment variable "`OTCLIB_HTTPLOGCHANNEL`" to the name of the log channel to record the information on. The environment variable needs to be set prior to the first request being received by the application through any instance of the `HttpDaemon` class.

```
dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)

netsvc.mergeEnviron("OTCLIB_HTTPLOGCHANNEL","")

daemon = netsvc.HttpDaemon(8000)
filesrvr = netsvc.FileServer(os.getcwd())
daemon.attach("/",filesrvr)
daemon.start()

dispatcher.run()
```

The format of the logged messages is the same as Apache web server common log file format except that no matter what version of HTTP is used, the url component of the request is always expanded to its complete form. That is, it will be prefixed with "`http://hostname:port`" as appropriate. Normally this would only be the case if the request originated with a client supporting HTTP/1.1 protocol and a full url had been supplied by the client.

If you do not want information about requests appearing in the default log file, but want to split out the logged messages into a distinct log file, or otherwise treat them in a special way, use a hidden log channel and create a user defined log channel to capture them.

# Servlet Objects

To make the most of the HTTP servlet framework it will be necessary to create your own servlets for interacting with your application. Servlets can be written to handle basic requests against a resource, or requests where form data is supplied. Special purpose servlets which process arbitrary content associated with a request may also be created. Having created a servlet, it can be integrated into an application by defining a custom HTTP server object, or by storing it as a file and using a plugin, in association with the file server object.

As the HTTP servlet framework is implemented on top of an event system and doesn't rely upon threads, it is necessary to be mindful of how servlets are implemented to avoid a situation where the code blocks. If the code does block it will effectively stop the whole application. The event system should therefore be used as appropriate where concurrency is required. If communication with service objects in a remote process is required to obtain data to satisfy a request, this will be essential.

## Processing a Request

In order to implement your own HTTP servlet, you need to create a new class which derives from the `HttpServlet` class. If the request your HTTP servlet is to handle does not have any content associated with it, you will only need to override the "`processRequest()`" member function.

The "`processRequest()`" member function will be called immediately after the HTTP server object has returned a valid HTTP servlet. If the HTTP servlet doesn't need to process any content associated with a request, it will typically be able to generate a response straight away and the servlet can then be destroyed.

```
class HttpServlet(netsvc.HttpServlet):
  def processRequest(self):
```

```
if self.requestMethod() != "GET":
  self.sendError(400)
else:
  self.sendResponse(200)
  self.sendHeader("Content-Type","text/plain")
  self.endHeaders()
  self.sendContent("Hi there.")
  self.endContent()
```

The major member functions of the HTTP servlet class used to interrogate the details of the HTTP request are "`requestMethod()`", "`requestPath()`" and "`queryString()`". It is these methods you would use to determine what the HTTP servlet is to do. It may be the case however that it is only necessary to validate the type of request method. This would occur where the HTTP server object had already identified a resource against which a request was being made and supplied the handle for that resource when the HTTP servlet was created.

Having determined the validity or otherwise of a request, the HTTP servlet can do a number of things. In the event of an error the HTTP servlet can use the "`sendError()`" member function to generate an error response. The first argument to "`sendError()`" should be the appropriate HTTP response code. An optional second argument may also be supplied consisting of valid HTML text. This text will be included within the body of the HTML document generated by the "`sendError()`" member function.

If the request is valid, the HTTP servlet might instead generate its own response including any appropriate content. To start the response the "`sendResponse()`" member function must be called. The first argument to "`sendResponse()`" would typically be be "`200`", indicating a successful response. A HTTP servlet may if it wishes supply any valid HTTP response code here. In fact, the "`sendError()`" member function is merely a shorthand method for generating an error response and underneath actually uses the same functions as described here.

The HTTP servlet may now include any HTTP headers by calling "`sendHeader()`". The arguments to "`sendHeader()`" should be the name of the header and its string value. Whether or not any HTTP headers are included, the member function "`endHeaders()`" must now be called.

To include content in a response the "`sendContent()`" member function is used. This may be called multiple times. When all content has been sent, the "`endContent()`" member function should be called. Calling the "`endContent()`" member function will have the affect of closing off the response and once the "`processRequest()`" member function returns, the servlet will able to be destroyed.

## Persistent Connections

A persistent connection is one whereby the connection to the client can be maintained after a response has been sent. This allows a HTTP client to submit additional requests without the need to create a new

connection. Negotiation of persistent connections between the HTTP client and server is managed by using special HTTP request and response headers.

Where possible the session manager will undertake to maintain persistent connections without you needing to take any special actions. This is done as a result of the session manager inserting on your behalf the special headers as appropriate when you call the "`endHeaders()`" member function.

If the client has requested a persistent connection and supplied a valid content length in the request headers, and you include a valid content length header in the response headers, the session manager will aim to maintain the connection. If you do not include a valid content length header in the response headers, or "`sendError()`" was used to generate a response, the connection will always be shut-down.

Note that when a HTTP client does send an additional request over the same connection, it will not be the same HTTP servlet instance that handles the request. Each request received will always be separately parsed, with the appropriate HTTP server object and servlet used each time.

## Delaying a Response

The servlet framework is implemented on top of the event system. As a result, it is not mandatory that a complete response be generated by the "`processRequest()`" member function. Instead, the servlet could execute some action which would result in a callback at a later point in time. When that callback occurs, then it might complete the response.

```
class HttpServlet(netsvc.HttpServlet,netsvc.Agent):
  def __init__(self,session):
    netsvc.HttpServlet.__init__(self,session)
    netsvc.Agent.__init__(self)
  def processRequest(self):
    if self.requestMethod() != "GET":
      self.sendError(400)
    else:
      self.sendResponse(200)
      self.sendHeader("Content-Type","text/plain")
      self.endHeaders()
      self.startTimer(self.completeResponse,10,"timeout")
  def completeResponse(self,tag):
    self.sendContent("Hi there.")
    self.endContent()
```

This is useful where the servlet needs to wait until data needed to formulate a response is available or where some form of time dependent server push mechanism is being implemented. Note however that special steps may be required in these situations to cope with a HTTP client prematurely closing the connection.

## Destruction of Servlets

The destruction of a servlet can come about as a result of two situations. The first situation is where a servlet handles a requests and generates a response, whether that be successful or otherwise. The second situation is where the HTTP client closes the connection before the servlet has sent a complete response.

The fact that the actions of a servlet may need to be aborted before it has finished complicate the destruction of a servlet. This is because any callback which may have been set up will result in a reference count against the servlet object. The existance of such references will actually prevent the immediate destruction of the servlet object. If that reference is never deleted, the servlet object may never be destroyed.

All this means that it isn't sufficient for the servlet framework to delete its own reference to an instance of a HTTP servlet and expect that it will be destroyed. Instead, it is necessary to introduce a special member function to the `HttpServlet` class and require that any derived class extend it as appropriate to cancel any callbacks or otherwise cause external or circular references to the servlet to be deleted.

The name of this member function is "`destroyServlet()`". The member function will be called when a HTTP client prematurely closes the connection. So that only one mechanism is employed to ensure a servlet is destroyed, the member function is also called subsequent to a servlet generating a complete response.

```
class HttpServlet(netsvc.HttpServlet,netsvc.Agent):
  def __init__(self,session):
    netsvc.HttpServlet.__init__(self,session)
    netsvc.Agent.__init__(self)
  def processRequest(self):
    if self.requestMethod() != "GET":
      self.sendError(400)
    else:
      self.sendResponse(200)
      self.sendHeader("Content-Type","text/plain")
      self.endHeaders()
      self.startTimer(self.completeResponse,10,"timeout")
  def completeResponse(self,tag):
    self.sendContent("Hi there.")
    self.endContent()
  def destroyServlet(self):
    netsvc.HttpServlet.destroyServlet(self)
    netsvc.Agent.destroyReferences(self)
```

The first action of the derived version of the member function "`destroyServlet()`" should be to call the base class version of the function in the `HttpServlet` class. The member function should then do what is ever necessary to ensure that references to the servlet are deleted. If the servlet had

been derived from the `Agent` class, this would include calling the "`destroyReferences()`" member function.

## Processing Content

If the function of a HTTP servlet entails that the content associated with a request be processed in some way, it will be necessary to override the "`processContent()`" member function. The "`processContent()`" member function will only be called subsequent to "`processRequest()`" being called, and only provided that "`processRequest()`" hadn't already dealt with the request and sent a complete response.

As the means to determine how much content to expect is dependent on the specifics of a request, no attempt is made to first accumulate the content into one block. Instead, the "`processContent()`" member function will be called multiple times if appropriate, once for each block of data which is read in. It is up to the "`processContent()`" member function to accumulate the data or otherwise process it, until it determines that all content has been received.

Typically, how much content is expected will be dictated by the presence of a HTTP content length header, or by a MIME multipart message boundary string as specificed in a HTTP content type header. Either way, it is up to the specific implementation of a HTTP servlet to know what to expect and deal with it appropriately.

```
class FormServlet(netsvc.HttpServlet):
  def processRequest(self):
    if self.requestMethod() not in ["GET","POST"]:
      self.sendError(501,"Request method type is not supported.")
    elif self.requestMethod() == "POST" \
        and self.contentLength() < 0:
      self.sendError(400,"Content length required for POST.")
    elif self.requestMethod() == "GET":
      self._environ = {}
      self._content = []
      self._contentLength = 0
      self._headers = self.headers()
      self._environ["REQUEST_METHOD"] = self.requestMethod()
      self._environ["QUERY_STRING"] = self.queryString()
      self._headers["content-type"] = \
          "application/x-www-form-urlencoded"
      try:
        form = cgi.FieldStorage(headers=self._headers, \
            environ=self._environ,keep_blank_values=1)
        self.processForm(form)
      except:
        netsvc.logException()
        self.shutdown()
    elif self.contentLength() == 0:
      self.processContent("")
```

```
  def processContent(self,content):
    self._content.append(content)
    self._contentLength = self._contentLength + len(content)
    if self._contentLength >= self.contentLength():
      self._environ["REQUEST_METHOD"] = self.requestMethod()
      self._content = string.join(self._content,"")
      self._content = self._content[:self.contentLength()]
      fp = StringIO.StringIO(self._content)
      try:
        form = cgi.FieldStorage(headers=self._headers, \
            environ=self._environ,keep_blank_values=1,fp=fp)
        self.processForm(form)
      except:
        netsvc.logException()
        self.shutdown()
  def processForm(self,form):
    self.sendResponse(501)
```

Member functions which a HTTP servlet may find useful here are "`contentLength()`" and "`contentType()`". The "`contentLength()`" member function returns an integer value corresponding to that defined by the HTTP content length header, or "`-1`" if no such field was provided. The "`contentType()`" member function returns the HTTP content type header. Note that this will include any supplied parameters so you will need to extract these yourself.

A HTTP servlet may also interrogate arbitrary headers using the member functions "`containsHeader()`" and "`header()`". These respectively indicate if a header exists and return its value. The name of a header should always be given as a lower case string. All headers may be obtained as a Python dictionary using "`headers()`".

If a HTTP servlet encounters an internal error at any time, it may call the "`shutdown()`" member function to abort all processing of the request. This will cause the connection to the HTTP client to be closed immediately, discarding any data which hadn't yet been sent. The instance of the HTTP servlet will then subsequently be destroyed.

## The Form Servlet

As processing of form data will be a common situation, an implementation of a form servlet is provided. This is called `FormServlet`. The implementation of this servlet is similar to the previous example except that it does additional processing to translate data from the types used by the `FieldStorage` class into standard Python lists and dictionaries. The name of the member function which you need to override to process the form is "`handleRequest()`".

```
class LoginServlet(netsvc.FormServlet):
  def handleRequest(self):
    if self.containsField("user") and \
        self.containsField("password"):
      user = self.field("user")
```

```
      password = self.field("password")
      if self.authenticateUser(user,password):
        self.sendResponse(netsvc.REDIRECT_TEMPORARY)
        self.sendHeader("Location",self.serverRoot())
        self.endHeaders()
        self.endContent()
      else:
        self.sendError(400)
    else:
      self.sendError(400)
  def authenticateUser(self,user,password):
    # ...
```

The existance of a field can be determined by calling the "`containsField()`" member function. The member function "`field()`" can then be called to retrieve the value for the field. All fields which have been set can be obtained as a dictionary using the "`fields()`" member function.

## Slow HTTP Clients

The HTTP servlet framework does not use multithreading but is layered on top of an event system. This fact means that it is not possible for a HTTP servlet to block, as doing so would block the whole process and stop anything else from running. For this reason, a HTTP servlet does not have direct access to the socket connection associated with a HTTP client. Instead, a HTTP servlet in sending data back to a HTTP client is effectively queueing the data for deliverly.

If the HTTP client is slow in reading data from a socket connection, the server side of the socket connection could effectively block. The underlying framework used to manage a socket connection will detect this, and will only send data over a socket connection when such a condition would not occur. A consequence of the queuing mechanism however is that any data will first be added to a queue and will only be sent after the servlet has returned.

For a small response this would not be a problem, but if the content associated with a response is large, the size of the process would grow dramatically if all data is queued at once. To avoid this, it is important that if sending large responses that they be sent in parts. Further, a HTTP servlet should suspend sending of further data when the socket connection would block, as this would again only serve to grow the amount of queued data and thus the size of the process.

To monitor changes in the state of the socket connection, a HTTP servlet should call the member function "`monitorCongestion()`", passing a callback function. The callback function supplied will be called when writing data to a socket connection would effectively block and also subsequently when the socket has cleared. These changes in state can be used to suspend and subsequently resume sending of data.

```
class TestServlet(netsvc.FormServlet):
  def __init__(self,session):
    netsvc.FormServlet.__init__(self,session)
```

```
    self._batch = None
    self._total = None
    self._count = 0
    self._job = netsvc.Job(self.generateContent)
  def destroyServlet(self):
    FormServlet.destroyServlet(self)
    self._job.cancel()
    self._job = None
  def handleRequest(self):
    if not self.containsField("batch") or \
        not self.containsField("total"):
      self.sendError(400)
    else:
      try:
        self._batch = int(self.field("batch"))
        self._total = int(self.field("total"))
      except:
        self.sendError(400)
      else:
        self.sendResponse(200)
        self.sendHeader("Content-Type","text/plain")
        self.endHeaders()
        self.monitorCongestion(self.clientCongestion)
        self._job.schedule(netsvc.IDLE_JOB)
  def generateContent(self):
    content = []
    for i in range(0,self._batch):
      self._count = self._count + 1
      content.append(string.zfill(self._count,60))
    content.append("")
    self.sendContent(string.join(content,"\n"))
    self._total = self._total - 1
    if self._total <= 0:
      self.ignoreCongestion()
      self.endContent()
    else:
      self.flushContent()
      self._job.schedule(netsvc.IDLE_JOB)
  def clientCongestion(self,status,pending):
    if status == netsvc.CONNECTION_CLEARED:
      self._job.reset()
      self._job.schedule(netsvc.IDLE_JOB)
    elif status == netsvc.CONNECTION_BLOCKED:
      self._job.cancel()
```

When a HTTP servlet no longer wishes to monitor the status of the socket connection the member function "`ignoreCongestion()`" can be called. Although not absolutely necessary, it is good practice to always call this just prior to calling the member function "`endContent()`" to close off the response.

Note that the Python wrapper around the C++ implementation of the HTTP servlet class performs buffering of content and will only pass content onto the C++ implementation when a set amount has been exceeded or the end of content has been indicated. If you suspend sending of further data, so that a HTTP client will see content produced so far, you may wish to flush out any buffered data by calling the "`flushContent()`" member function.

# Servlet Plugins

When using a file server object with the HTTP servlet framework, it is possible to associate a special purpose handler or plugin with requests against files with a particular extension. When a request is made against such a file, the plugin is used as an intermediary for the creation of a servlet to handle that request. The plugin can return a servlet which was loaded into the application at startup, or might also load the servlet from the file or otherwise generate a servlet on the fly.

This feature means that the functionality of an application can to a degree be extended but without the need to have such functionality hardwired into the application itself. The functionality of an application might even be extended or reduced at run time by the simple act of adding or removing files from the file system. This eliminates the need to restart an application everytime a change is required.

## Python Plugin

To support implementations of HTTP servlets being contained within files residing in the file system, as opposed to being hardwired into the application itself, the `PythonPlugin` class is provided. This gives greater flexibility as it would not be necessary to restart the application to add in new functionality. The plugin can also detect when a servlet file has been modified and automatically reload it as necessary.

So that the Python import mechanism can find these files, they should be given a "`.py`" extension. This mapping is however not built in and it is necessary to register the extension as being associated with the particular plugin in question.

```
filesrvr = netsvc.FileServer(os.getcwd())
filesrvr.plugin(".py",netsvc.PythonPlugin())
```

The effect of this registration will be that whenever a file with extension "`.py`" is requested by a HTTP client, the plugin object will be executed as a callable object, with the HTTP session object and the name of the file being passed as arguments. In this case, the `PythonPlugin` object will import the file as if it is a Python module, obtain from it a reference to the HTTP servlet and then create an instance of the HTTP servlet to service the request.

The main difference when writing a servlet to be contained in a file and loaded in this way, as opposed to one which is hardwired into the actual application, is that it is necessary to provide a hook for creating an instance of the servlet. This is done by providing a definition within the file of the symbol "`__servlet__`".

```
import netsvc

class HttpServlet(netsvc.HttpServlet):
  def processRequest(self):
    if self.requestMethod() != "GET":
      self.sendError(400)
    else:
      self.sendResponse(200)
      self.sendHeader("Content-Type","text/plain")
      self.endHeaders()
      self.sendContent("Hi there.")
      self.endContent()

__servlet__ = HttpServlet
```

In the simplest case, the symbol "`__servlet__`" can be defined to be a reference to the actual servlet type. The `PythonPlugin` object will execute "`__servlet__`" with the expectation it is a callable object, supplying it with a single argument of the HTTP session object. In the above case this will immediately result in an instance of the servlet being created.

An alternative might be that "`__servlet__`" be defined as a function. This would allow one of a number of servlets to be chosen based on specific criteria, such as the time of day or whether the service is operational.

```
def __servlet__():
  return HttpServlet
```

If a servlet requires additional arguments to be supplied along with the HTTP session object, a proxy object could instead be defined which transparently supplies the additional arguments. For example, a servlet designed to facilitate directory browsing might be supplied the name of the directory in which the servlet file resided, with the servlet generating a directory listing of files contained in the directory.

```
class ServletProxy:
  def __call__(self,session):
    directory = os.path.dirname(__file__)
    return BrowseServlet(session,directory)
```

```
__servlet__ = ServletProxy()
```

Note that an instance of the servlet is created for each request. That is, unlike other similar systems available for Python, an instance of a servlet object is not cached and reused. If you need to maintain state between requests, such information should be factored out into a distinct service agent object.

## Module Caching

In the case of the `PythonPlugin` object, when the file containing the actual servlet is read in, it is compiled into Python byte code and cached. This means that a subsequent request against that servlet will use the cached byte code and will not reread and recompile the file. So that is isn't necessary to stop and start the application if the file is changed, upon each subsequent request a check is made to see if the file has since been modified. If the file has been modified, it will be reread and recompiled ensuring that changes made to the file are visible.

Note that this mechanism will only detect if the actual servlet file has been modified. If that servlet file imports other modules using the Python "`import`" command and it is those other modules which have been changed, the cached servlet will still be used. This is acceptable where the other modules contain core program logic on which other parts of the application are dependent, but not in the case where the separate module contains a servlet base class defining the structure of a web page and it is the structure of the web page which you wish to change.

To cater for this situation, a special mechanism is provided for importing of modules which define servlet base classes or functionality related to the presentation of a web page. When this mechanism is used, that the servlet file is dependent on the module is recorded and a servlet file will be reread and recompiled, as will the module it depends on, when only the module had changed.

```
import os
import netsvc

cache = netsvc.ModuleCache()
directory = os.path.dirname(__file__)
_template = cache.importModule("_template",directory)

class HttpServlet(_template.PageServlet):
  def writeContent(self):
    self.writeln("Hi there.")

__servlet__ = HttpServlet
```

This means that the structure of a page can be defined in a common place, with each servlet file only defining the content specific to that page. The module caching mechanism should however only be used for this purpose. It is also recommended that for a particular module file, you not mix this mechanism and the standard Python import system, but use this system exclusively.

Note that a module imported in this way can use the same mechanism to import further modules with the dependence on those additional modules also being considered when the initial file is requested. Be aware however, that if files are located on a different machine to that which the application is running on and the clocks are not sychronised properly, updates may not always be detected correctly.

## Writing a Plugin

A plugin can be any callable object, so long as it accepts as arguments when called, a HTTP session object and the name of a file which is the target of the request. The plugin may therefore be a type, a function, or an object which overrides the "`__call__`" method. Which approach is used will depend on whether state needs to be preserved between invocations of the plugin. If no state needs to be preserved, a simple function may be the most approrpiate.

```
def factory(session,file):
  return netsvc.FileServlet(session,file)

filesrvr.plugin(".txt",factory)
```

If a HTTP servlet when being constructed takes the same arguments as those passed to the plugin, ie., the HTTP session object and the name of a file, the servlet itself might instead be registered as the plugin.

```
filesrvr.plugin(".txt",netsvc.FileServlet)
```

Where state needs to be preserved, registration of an actual object instance which can hold the state, may be a better approach.

```
class Plugin:
  def __call__(self,session,file):
    return netsvc.FileServlet(session,file)

filesrvr.plugin(".txt",Plugin())
```

## Plugin Aliasing

When a plugin is registered, the filename extension specified must appear in the URL used by the HTTP client when accessing that resource. This has the perhaps unwanted effect of exposing details about how the web pages are implemented. This may limit to what extent you can easily change the implementation later on, but may also give a malicious user ideas about how they may remotely break into your system.

For these reasons, although a servlet file might be required to use the extension "`.py`", if that servlet file always produces HTML, it may be preferable that that resource always be accessed by using a "`.html`" extension. An ability to do this can also be useful in the case where a resource is initially stored as a static file with a "`.html`" extension, but is later changed to be dynamically generated using

a servlet. In this later case, the name of the resource can remain the same, and no references to the resource need to be changed.

To facilitate use of an alias, an optional argument can be supplied to the "`plugin()`" member function defining the alternate extension the resource should be identified with. If you wanted all servlet files accessible using a particular instance of a file server object to be accessed using a "`.html`" extension instead of the "`.py`" extension, the string "`.html`" would be supplied as the optional third argument to the "`plugin()`" member function.

```
filesrvr.plugin(".py",netsvc.PythonPlugin(),".html")
filesrvr.hide(".py")
```

Note that if the "`hide()`" member function isn't also called with the "`.py`" extension, the servlet file would still be accessible with a "`.html`" extension, but a request against the "`.py`" extension would yield the actual Python source code. This would not be an issue if the plugin had at the same time also been registered for "`.py`" files, but without the alias.

If the "`.py`" file is hidden, if the servlet file was called "`login.py`", it would be accessable as "`login.html`", but an attempt to use "`login.py`" would result in a HTTP error response indicating that the file could not be found. If the "`.py`" file isn't hidden, but the plugin is registered twice, once without an alias and once with the alias "`.html`", both "`login.py`" and "`login.html`" would work.

If the servlet files are providing the roles of CGI scripts, it may be desirable for the files to use no extension at all. That is, the file should be accessed as "`login`" instead of "`login.py`". If this is the case, rather than "`.html`", an empty string can be provided.

```
filesrvr.plugin(".py",netsvc.PythonPlugin(),"")
```

Be aware that the optional argument to "`plugin()`" defining the alias is actually treated as a filename suffix and not strictly as an extension. What this means is that that argument need not start with "`.`", but can be any arbitrary string in which the name of a resource ends. This means it is actually possible to synthesis new resources as long as they derive from an actual file.

One use of this is a plugin which returns a servlet which generates a thumbnail version of an image. For example, if an image file was originally called "`holiday.gif`", a request against "`holiday-thumbnail.gif`" could me made to generate a thumbnail image on the fly.

```
def factory(session,file):
  return ThumbnailServlet(session,file)

filesrvr.plugin(".gif",factory,"-thumbnail.gif")
```

# Remote Access

The service agent and message exchange framework operate based on the concept of processes which are a part of a distributed application being permanently connected together. This model works fine on corporate networks, but is not always practical when run across the Internet. One drawback of this approach is that it is often necessary to open up special ports on a corporate firewall to permit access.

For many instances where communication across the Internet is required, a connected model of operation isn't actually required. Instead, many types of operations can be carried out using a request/reply model whereby a connection is only maintained for the lifetime of the request. This is precisely the type of mechanism which is used by HTTP.

Because of the wide acceptance for HTTP a number of remote procedure call protocols have been developed which operate within the bounds of a HTTP request. The most well known of these are XML-RPC and SOAP. Unfortunately, both of these protocols are actually lacking in certain respects and have not been found to be a totally satisfactory medium.

In place of these protocols, an alternative RPC over HTTP protocol is provided called NET-RPC. At present, the only client available is implemented using Python. If you are writing a closed system this shouldn't present a problem. In those cases where public access may be required, gateways for XML-RPC and SOAP are still available, but using them will place a limitation on the type of data which you can pass around.

Note that whichever RPC over HTTP protocol you do decide to use, the code for your services is the same. In fact, your application may include gateways for all three protocols and a user can use whichever type of client they find easiest. In this respect, it doesn't matter too much which protocol wins out.

Even if a new protocol comes along, it is a relatively simple matter to incorporate yet another gateway, again without you having to make modifications to the core of your system.

## The RPC Gateway

The gateway which accepts an RPC request is actually an instance of a HTTP server object. The gateway will accept a request and based on the URL determine which service the request applies to. The request will then be translated into a call over the service agent framework, with the corresponding result being packaged up and returned to the remote client.

Because the service agent framework can operate in a distributed manner using the message exchange framework, the service which a request applies to need not even be in the same process as the RPC gateway. So that a remote client can't access any arbitrary service however, a mechanism is provided to limit which services are actually visible. The mechanisms for client and user authorisation implemented by the HTTP servlet framework can also be used to block access as appropriate.

For the NET-RPC protocol, the RPC gateway is implemented by the `RpcGateway` class. When created, the gateway needs to be supplied the name of a service group. Only those services which are a member of that service group will be accessible through that particular instance of the RPC gateway. Having created an instance of the RPC gateway, it needs to be mapped into the URL namespace of a HTTP daemon object.

```
import netsvc
import signal

class Validator(netsvc.Service):
  def __init__(self,name="validator"):
    netsvc.Service.__init__(self,name)
    self.joinGroup("web-services")
    self.exportMethod(self.echo)
  def echo(self,*args)
    return args

dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)

validator = Validator()

port = 8000
group = "web-services"
httpd = netsvc.HttpDaemon(port)
rpcgw = netsvc.RpcGateway(group)
httpd.attach("/service",rpcgw)
httpd.start()

dispatcher.run()
```

In this example, any HTTP request made using a URL whose path falls under the base URL of "`http://localhost:8000/service/`", will be regarded as being a NET-RPC request. The name of the service which a request applies to is determined by removing the base URL component from the full URL. The full URL used to access the service in this example would therefore be "`http://localhost:8000/service/validator`". Note that the service is only visible however, because it had added itself to the group "web-services", the same group as the RPC gateway had been initialised with.

The methods of the service which are available are the same as those which would be accessible over the service agent framework internal to your application. That is, a service must export a method for it to be accessible. The only such method available in this example would be "`echo()`".

## The Client Application

Client side access to the NET-RPC protocol is available through the Python "`netrpc`" module. This module is not dependent on the "`netsvc`" module and is pure Python. The name of the class used to make a request to a remote service is `RemoteService`. This class behaves in a similar fashion to the `LocalService` class from the "`netsvc`" module except that the service name is replaced with the URL identifying the remote service.

```
import netrpc

url = "http://localhost:8000/service/validator"
service = netrpc.RemoteService(url)
print service.echo(1,1L,1.1,"1")
```

Only the "http" protocol is supported. If the URL specifies an unsupported protocol, the exception `AddressInvalid` will be raised. If the URL didn't identify a valid service on the remote host, a `ServiceUnavailable` exception is raised. Other possible exceptions which may be raised are `AuthenticationFailure` and `TransportFailure`. All the more specific exceptions actually derive from `ServiceFailure` and the `ServiceFailure` exception is also used for errors generated by the service itself, so it is often sufficient to watch out for just that type of exception.

## Restricting Client Access

In addition to being able to dictate precisely which services are visible, it is also possible to restrict access to specific clients. This can be done by allowing only certain hosts access, or by limiting access to specific individuals by using user authentication. Both schemes rely on features within the existing HTTP servlet framework.

```
class HttpDaemon(netsvc.HttpDaemon):
  def __init__(self,port,hosts=["127.0.0.1"]):
    netsvc.HttpDaemon.__init__(self,port)
    self._allow = hosts
  def authorise(self,host):
```

```
        return host in self._allow:

    class RpcGateway(netsvc.RpcGateway):
      def __init__(self,group,users=None):
        netsvc.RpcGateway.__init__(self,group):
        self._allow = users
      def authorise(self,login,password):
        return self._allow == None or \
            (self._allow.has_key(login) and \
            self._allow[login] == password)

    users = { "admin": "secret" }

    port = 8000
    group = "web-services"
    httpd = HttpDaemon(port)
    rpcgw = RpcGateway(group,users)
    httpd.attach("/service",rpcgw)
    httpd.start()
```

When user authentication is being used, the login and password of the user can be supplied as additional arguments to the `RemoteService` class when it is created.

```
url = "http://localhost:8000/service/validator"
service = netrpc.RemoteService(url,"admin","secret")
print service.echo(1,1L,1.1,"1")
```

If a login and password aren't supplied when required, or the details are wrong, the `AuthenticationFailure` exception will be raised.

## Duplicate Services

Because a URL identifies a unique resource, a conflict arises due to the fact that within the service agent framework it is possible to create multiple services with the same name. What happens in this circumstance is that the RPC gateway will remember which service agent was the first it saw in the required service group, having a particular service name. While that particular service agent exists, it will always use that service agent as the target of requests.

When there are multiple service agents with the same service name and the first one seen by the RPC gateway is destroyed, the RPC gateway will then fall back to using the second one it saw. That is, the RPC gateway will always use the service agent which it has known about the longest. In general, if you intend to make services accessible using the RPC gateway, it is recommended that you always use unique service names within the service group dictating which services are actually visible.

## User Defined Types

The NET-RPC protocol supports all the types supported by the service agent framework, as well as the concept of user defined scalar types. That is, if a service responds with data incorporating additional scalar types, they will by default be passed back as instances of the `Opaque` type, where the "`type`" attribute gives the name of the type and the "`data`" attribute the encoded value. Similarly, new types may be sent by initialising an instance of the `Opaque` type with the name of the type and the value in its encoded form.

```
url = "http://localhost:8000/service/validator"
service = netrpc.RemoteService(url,"admin","secret")

# following are equivalent
value = complex(1,1)
print service.echo(value)
print service.echo(netrpc.Opaque("python:complex",repr(value)))
```

Encoders and decoders for additional user defined scalar types can be provided by registering the appropriate functions using the "`encoder()`" and "`decoder()`" functions available in the "`netrpc`" module. The functions for registering the encoder and decoder functions are used in exactly the same was as those in the "`netsvc`" module. In fact, they are the same functions as the "`netsvc`" module imports them from the "`netrpc`" module, as it does for the implementations of all of the extended types.

As is the case in the service agent framework, you need to be mindful about the effect of registering arbitrary encoders and decoders at global scope, especially if your client application makes calls against different services implementing their own scalar types. This becomes even more of an issue if the "`netrpc`" module is used to make client side calls from inside a server side application using the "`netsvc`" module. This is because they will share the same global encoders and decoders.

If you need to support types which are specific to a service being called, rather than registering the encoder and decoder function at global scope, the safer way is to supply your own functions just for that service. This is done by supplying the function using a keyword argument when initialising the instance of the `RemoteService` class. The keyword argument for the encoder function is "`encode`" and that for the decoder function is "`decode`". The functions you supply should call the corresponding global function if it doesn't know what to do with a specific type.

```
def encodeObject(object):
  if type(object) == MySQLdb.DateTimeType:
    return ("xsd:string",object.strftime())
  elif type(object) == MySQLdb.DateTimeDeltaType:
    return ("xsd:string",str(object))
  return netsvc.encodeObject(object)

url = "http://localhost:8000/service/validator"
service = netrpc.RemoteService(url,encode=encodeObject)
```

## Managing User Sessions

A common practice with web based services is to have a request initiate a unique session for a user. Having opened the session, any requests will then be identified with that session, with information regarding the session potentially being cached on the server side until the session is closed. Such a session might also be used as a way of allocating a server side resource to that user, or creating a database cursor dedicated to a particular user so more complex queries can be made.

A scheme suitable for use over the service agent framework was previously described, however that implementation was based on the ability to subscribe to the existence of the owner of the session, with the session being automatically closed when the owner was destroyed. When the RPC gateway is used, this approach can't be used, as the sender of the request will be a transient service created by the RPC gateway to service just that request. An alternative when the RPC gateway is being used is to automatically close the session after a set period of inactivity.

```
class Database(netsvc.Service):
  def __init__(self,name="database",**kw):
    netsvc.Service.__init__(self,name)
    self._name = name
    self.joinGroup("database-services")
    self._database = MySQLdb.connect(**kw)
    self._cursors = 0
    self.exportMethod(self.cursor)
  def executeMethod(self,name,method,params):
    try:
      return netsvc.Service.executeMethod(self,name,method,params)
    except MySQLdb.ProgrammingError,exception:
      self.abortResponse(1,"Programming Error","db",str(exception))
    except MySQLdb.Error,(error,description):
      self.abortResponse(error,description,"mysql")
  def cursor(self,timeout=60):
    self._cursors = self._cursors + 1
    name = "%s/%d" % (self._name,self._cursors)
    cursor = self._database.cursor()
    Cursor(name,cursor,timeout)
    child = "%d" % self._cursors
    return child
```

The idea is that when a request is made, a unique instance of a service is created specific to the session, with a name which is then passed back to the remote client. In the example shown, if the service was originally accessible using the URL "`http://localhost/database`", the instance of a service created for that specific session would be the same URL but with the session id appended, separated by "`/`". Eg., "`http://localhost/database/1`". Obviously, a session id which could not be easily guessed should however be used.

The client would now direct all future requests to the new URL. When the client has finished with the service it would call the "`close()`" method on the service. If for some reason the client did not ex-

plicitly close off the session, it would be automatically closed after a period of 60 seconds of inactivity, or whatever period was defined when the session was initiated. An implementation of the database cursor service for this example might be as follows.

```
class Cursor(netsvc.Service):
  def __init__(self,name,cursor,timeout):
    netsvc.Service.__init__(self,name)
    self.joinGroup("database-services")
    self._cursor = cursor
    self._timeout = timeout
    self._restart()
    self.exportMethod(self.execute)
    self.exportMethod(self.executemany)
    self.exportMethod(self.description)
    self.exportMethod(self.rowcount)
    self.exportMethod(self.fetchone)
    self.exportMethod(self.fetchmany)
    self.exportMethod(self.fetchall)
    self.exportMethod(self.arraysize)
    self.exportMethod(self.close)
  def encodeObject(self,object):
    if hasattr(MySQLdb,"DateTime"):
      if type(object) == MySQLdb.DateTimeType:
        return ("xsd:string",object.strftime())
      elif type(object) == MySQLdb.DateTimeDeltaType:
        return ("xsd:string",str(object))
    return netsvc.Service.encodeObject(self,object)
  def executeMethod(self,name,method,params):
    try:
      return netsvc.Service.executeMethod(self,name,method,params)
    except MySQLdb.ProgrammingError,exception:
      self.abortResponse(1,"Programming Error","db",str(exception))
    except MySQLdb.Error,(error,description):
      self.abortResponse(error,description,"mysql")
  def _restart(self):
    self.cancelTimer("idle")
    self.startTimer(self._expire,self._timeout,"idle")
  def _expire(self,name):
    if name == "idle":
      self.close()
  def execute(self,query,args=None):
    result = self._cursor.execute(query,args)
    self._restart()
    return result

  # additional methods

  def close(self):
    self._cursor.close()
```

```
        self.cancelTimer("idle")
        self.destroyReferences()
        return 0
```

Using the "netrpc" module to access the service, a client might be coded as follows. In this case a separate cursor is created in relation to the queries made about each table in the database.

```
import netrpc

url = "http://localhost:8000/database"
service = netrpc.RemoteService(url)

tables = service.execute("show tables")

timout = 30
for entry in tables:
  table = entry[0]
  print "table: " + table
  name = service.cursor(30)
  print "cursor: " + url + "/" + name
  cursor = netrpc.RemoteService(url+"/"+name)
  cursor.execute("select * from "+table)
  desc = cursor.description()
  print "desc: " + str(desc)
  data = cursor.fetchall()
  print "data: " + str(data)
  cursor.close()
```

In general, giving open access to a database in this way may not be advisable, especially over the Internet. Such a mechanism might be restricted to a corporate intranet. Alternatively, custom interfaces should be layered on top of the database providing interfaces based on functional requirements.

## The XML-RPC Gateway

If the Python NET-RPC client implementation can’t be used because of the need to use a different language for the client, you might instead consider using the XML-RPC protocol. Clients for the XML-RPC protocol are available in many different languages, many of which are listed at "http://www.xmlrpc.com". The only change to your server application will be to instantiate an instance of the XML-RPC gateway instead of the NET-RPC gateway.

```
import netsvc
import netsvc.xmlrpc

dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)

validator = Validator()
```

```
port = 8000
group = "web-services"
httpd = netsvc.HttpDaemon(port)
rpcgw = netsvc.xmlrpc.RpcGateway(group)
httpd.attach("/service",rpcgw)
httpd.start()

dispatcher.run()
```

If you do decide to rely upon the XML-RPC protocol instead of the NET-RPC protocol, you will be constrained as to what types you can use. This is because the XML-RPC protocol has a more limited set of core types and is not type extendable as is the NET-RPC protocol. One major deficiency of the XML-RPC protocol, is that it has no way of passing a null value, such as that implemented by the Python `None` type. Some XML-RPC clients have been extended to support a null value, but this gateway does not implement such an extension.

A further complication which can arise in using XML-RPC is that the specification isn't precise in certain areas. Although an XML-RPC message is notionally XML, the specification indicates use of ASCII values in strings only. This is in conflict with XML which requires at least UTF-8. Another issue is that the XML-RPC specification mentions nothing about needing to support XML comments, CDATA or various other XML constructs. This has lead to some implementations of the protocol not supporting such features of XML and others relying on them.

Because of the inter operability issues which may arise due to the differences between different XML-RPC clients, a number of different XML-RPC protocol implementations are actually supported by the XML-RPC gateway. By default a pure Python implementation of routines for decoding and encoding XML-RPC messages is used. This implementation uses a full XML parser and should be able to handle anything XML dictates.

In general, when only small amounts of data is being passed back and forth, most of the cost of a remote procedure call is actually consumed in the costs of starting up and ripping down the TCP/IP connection by the client and of the server responding to the connection request. That the XML-RPC encoding and decoding routines are implemented in Python may not therefore have any significant impact. However, when large amounts of data are being passed around, this may not be the case.

An alternative to the Python implementation is one implemented in C++. This implementation will be quicker at decoding XML-RPC messages, but does not use a full XML parser and thus will not work if XML constructs such as comments and CDATA are used. The implementation also may not always handle out of order elements in the method call and struct elements. The C++ implementation if desired can be selected by supplying a keyword argument when initialising the XML-RPC gateway.

```
# Default uses Python implementation.
rpcgw1 = netsvc.xmlrpc.RpcGateway(group)

# Use C++ implementation instead.
rpcgw2 = netsvc.xmlrpc.RpcGateway(group,variant="c++")
```

```
# Explicitly specify use of Python implementation.
rpcgw3 = netsvc.xmlrpc.RpcGateway(group,variant="python")
```

Both these Python and C++ implementations of the routines for handling the XML-RPC protocol are supplied with OSE. Because of the limitations of the XML-RPC protocol in respect of passing a more diverse set of types, when these implementations are used, types which don't have a direct equivalent in XML-RPC will have their encoded value passed as a string, with a subsequent loss of type information.

For example, the Python `None` type will be sent as an empty string. Note this only applies to the result of a request when it is being returned via an XML-RPC request. Since XML-RPC doesn't support the extra types, a client strictly conforming to the XML-RPC protocol would not have been able to generate them in the first place.

Although there are numerous third party XML-RPC clients available, including a number for Python, an XML-RPC client is also provided with OSE. This client is interface compatible with that provided by the "`netrpc`" module and is available in the "`netrpc.xmlrpc`" module. This provides exactly the same interface as the "`netrpc`" module, even to the extent of being able to reconstruct the more informative failure responses provided by the service agent framework.

```
import netrpc.xmlrpc

url = "http://localhost:8000/service/validator"
service = netrpc.xmlrpc.RemoteService(url)
print service.echo(1,1L,1.1,"1")
```

What happens when a failure occurs is that the additional information provided by the service agent framework is encoded into the description field of an XML-RPC fault. When this is received by the "`xmlrpc`" module it extracts out the information into separate fields once more. If you are using a third party XML-RPC client this will not occur. What you will find instead is that the fault code will equate to the error code of a failure, with the description included with the fault looking something like the following.

```
origin -- the description

additional fault details
```

That is, the description is prefixed by the origin of the failure, separated by "`--`". The additional details of the failure will then appear separated from the description by a blank line. You could either use this as is, or separate out the information yourself.

Note that when using the "`xmlrpc`" module the encoders and decoders become largely irrelevant given that the XML-RPC protocol is not type extendable. Although the "`xmlrpc`" module provides an interface compatible with the "`netrpc`" module, it still may be used to make requests against third party XML-RPC servers.

## The SOAP Gateway

Yet another alternative to XML-RPC is the SOAP protocol. A starting point for SOAP is the site "http://www.develop.com/soap". Although SOAP is newer than XML-RPC and is notionally type extendable, it actually has some more significant limitations than XML-RPC. As with XML-RPC, it is only recommended that you use this protocol in preference to the NET-RPC protocol if you really have to. In doing so you will need to write your code keeping in mind these limitations.

The biggest limitation of the SOAP protocol at present is that the specification uses XML element names to describe the member keys in structures. At present, using the default SOAP encoding such structures are the only way of representing a Python dictionary. The service agent framework already restricts keys in dictionaries to strings, but the SOAP protocol limits what values those keys can have because of the XML naming rules.

More specifically, XML says that an element name, and thus a key in a Python dictionary, cannot start with a number. Further to this, a key would not be able to contain white space, a whole host of punctuation characters, nor would a key be able to start with the string "`xml`" in any mix of upper or lower case. It is also not possible to represent an empty dictionary in SOAP. These amount to being quite a severe restriction and effectively means that dictionaries need to be converted into a list of key/value tuples in order to be sent correctly.

The Apache SOAP toolkit has defined an alternative compound type which would be suitable for representing a dictionary, but only a few SOAP toolkits actually support it. It is also questionable as to whether a SOAP client would interchangeable accept this new type in any place where a structure may appear. Inevitably, Microsoft and IBM will come up with yet another scheme for doing the same thing, so any consensus is likely to be long in coming.

At present the default SOAP gateway relies on a third party "ZSI" module from the "`pywebsvcs`" toolkit. To use the SOAP gateway, you will need to have separately obtained this third party module from "http://sourceforge.net/projects/pywebsvcs" and installed it. You must have version 1.2 RC2 or later of this package. It is not possible to use version 1.1 or earlier of this package because of bugs in the package.

If you still wish to use the SOAP gateway, the only real change you would need to make to your code would be to instantiate an instance of the SOAP gateway in place of the NET-RPC gateway. Obviously though, if your service produces data which doesn’t fit within the limitations of the SOAP protocol the gateway will generate XML which a client will not be able to parse.

```
import netsvc
import netsvc.soap

dispatcher = netsvc.Dispatcher()
dispatcher.monitor(signal.SIGINT)

validator = Validator()
```

```
port = 8000
group = "web-services"
httpd = netsvc.HttpDaemon(port)
rpcgw = netsvc.soap.RpcGateway(group)
httpd.attach("/service",rpcgw)
httpd.start()

dispatcher.run()
```

To complement the SOAP gateway, a SOAP client is provided in the "`netrpc.soap`" module. This client is only suitable for use against SOAP based web services which rely on positional arguments. Most web services use WSDL and therefore require named parameters, making the client unsuitable in those cases.

```
import netrpc.soap

url = "http://localhost:8000/service/validator"
service = netrpc.soap.RemoteService(url)
print service.echo(1,1L,1.1,"1")
```

If using this client against a SOAP server written using a different system, it may be necessary to bind the method call to a specific namespace and/or provide a specific value for the "`SOAPAction`" header of the SOAP request. If this is the case, a method namespace can be supplied using the "`ns`" keyword argument when creating the instance of the `RemoteService` class. Similarly, a value for the "`SOAPAction`" header can be supplied using the "`soapaction`" keyword argument. If no "`soapaction`" argument is supplied, the value of the "`SOAPAction`" header will be a pair of double quotes.

```
url = "http://services.soaplite.com/hibye.cgi"
uri = "http://www.soaplite.com/Demo"
service = netrpc.soap.RemoteService(url,ns=uri)
print service.hi()
```

If a particular SOAP server requires a different method namespace or "`SOAPAction`" header for each method called, the "`ns`" and "`soapaction`" keyword arguments can instead be supplied at the point the call is made, rather than when the `RemoteService` object is created.

Note that although SOAP is type extendable, because the namespace associated with a new type name must be bound to a URI, the lesser described type information used by the service agent framework can’t be transparently translated into valid XML as per the SOAP encoding rules. You are therefore limited to types described by the XML Schema Datatypes specification, although at present not all such types may be translated by the SOAP gateway. In the future as experience and demand dictates, the gateway will be amended however to ensure that any types from the XML Schema Datatypes are passed through appropriately.

In respect of a failure response generated by the service agent framework, the four fields will be encoded as separate fields within the SOAP fault structure detail element enclosed with an XML element

called "`ServiceFailure`". All elements will be qualified in the OSE namespace. Inadequate prior art has been found as to the most appropriate way to make use of the detail element, so it may be necessary to change this in the future if necessary.

## Using Multiple Gateways

Because it is possible to attach multiple HTTP server objects to a particular instance of a HTTP daemon object, you aren't restricted to having only one instance of an RPC gateway. The first consequence of this fact is that for which ever protocol you intend to use, multiple RPC gateways can be created which map to distinct parts of the URL namespace.

Such RPC gateways can and would generally be associated with different groups of services. It is possible that some of the RPC gateways might be protected using user authentication. At the same time, a HTTP file server object or custom HTTP server object might also be attached to the same HTTP daemon.

```
files = netsvc.FileServer(os.getcwd())
httpd.attach("/download",files)

user = netsvc.RpcGateway("web-services")
httpd.attach("/service",user)

admin = netsvc.RpcGateway("admin-services")
httpd.attach("/admin",admin)
```

When creating RPC gateways, you also aren't restricted to them all being for the same protocol. As long as they are hosted under different parts of the URL namespace, gateways for all three of the RPC over HTTP protocols currently supported could be provided against the same set of services. In doing this, as long as your service fits within the lowest common denominator with respect to the limitations of the XML-RPC and SOAP protocols, you could leave it up to the user as to which protocol they want to use.

```
netrpcgw = netsvc.RpcGateway("web-services")
httpd.attach("/netrpc",netrpcgw)

xmlrpcgw = netsvc.xmlrpc.RpcGateway("web-services")
httpd.attach("/xmlrpc",xmlrpcgw)

soapgw = netsvc.soap.RpcGateway("web-services")
httpd.attach("/soap",soapgw)
```

Using the various features of the HTTP servlet framework, a diverse set of interfaces could be presented through the same HTTP daemon. In general though, it is recommended that a full blown HTTP server such as Apache still be used for performing as much of the web serving capabilities as possible.

A suitable model may be to use PHP or PSP on your main web server and have it make requests into the back end application using one of the RPC over HTTP protocols as necessary. This has the benefit

of also pushing a lot of the security issues onto the main web server where they are more often than not easier to manage and deal with. If it was necessary to expose some part of the application direct to outside users, one approach might be to make use of the Apache "`mod_proxy`" module rather than directly exposing the application.